Solving set partitioning problems using lagrangian relaxation

van Krieken, M.G.C.

Publication date:
2006

Document Version
Publisher's PDF, also known as Version of record

SOLVING SET PARTITIONING PROBLEMS USING LAGRANGIAN RELAXATION

SOLVING SET PARTITIONING PROBLEMS USING LAGRANGIAN RELAXATION

Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Tilburg,
op gezag van de rector magnificus, prof. dr. F.A. van der Duyn Schouten,
in het openbaar te verdedigen ten overstaan van
een door het college van promoties aangewezen commissie
in de aula van de Universiteit op

woensdag 15 maart 2006 om 16.15 door

Maria Gertruda Cornelia van Krieken

geboren op 3 januari 1979 te ‘s-Hertogenbosch.

Promotor:	Prof.dr.ir. H.A. Fleuren
Copromotor:	Dr.ir. M.J.P. Peeters

"*Science is like a blind man in a dark room, looking for a black hat that may not even be there.*"

Karl Popper

To Ieke

and my parents

Prologue

Although it is impossible for me to thank everybody who has supported or inspired me in the last four years, I would like to take this opportunity to thank the most important people who contributed to this thesis.

Obviously the first people to thank are my promotor, Hein Fleuren, and my copromotor, René Peeters. I thank them for the interesting discussions, inspiring ideas and valuable comments. Moreover, I am grateful for the opportunity Hein has given me to fulfil this research and the enthusiastic way in which he supported me during the last four years.

I would also like to thank my colleagues at CentER AR for the pleasant working environment, the coffee and lunch breaks and the daily walk through the woods. In particular, I want to thank my roommate Cindy for the good atmosphere and Ilse for her great support.

Last, I would like to thank my friends and family for their support. I especially want to mention my parents, who have always supported me an stimulated me to develop myself to the person I am today. Furthermore, very special thanks go to Ieke. The last seven years you have enhanced my personal as well as my professional life. I thank you for your valuable comments and for all the joint work I enjoyed very much. But most of all I thank you for your continuing love, patience and encouragement.

Maaike van Krieken
Tilburg, July 2005

Contents

Chapter 1

Introduction

This chapter introduces the set partitioning problem and its applications and defines the goals of the research discussed in this thesis.

1.1 The set partitioning problem

Given a collection of subsets of a certain root set and costs associated to these subsets, the set partitioning problem is the problem of finding a minimal cost partition of the root set. Formally, the set partitioning problem can be written as follows.

Min $\sum_{j \in J} c_j \cdot x_j$ [1.1]

Subject to

$\sum_{j \in J} a_{rj} \cdot x_j = 1 \qquad \forall r \in R$ [1.2]

$x_j \in \{0,1\} \qquad \forall j \in J$ [1.3]

Here, R is the set of constraints or rows of the problem (root set) and J is the collection of subsets or columns of the problem. The matrix $A = \{a_{rj}\}$ is defined such that $a_{rj} = 1$ if subset j contains row r and 0 otherwise. The costs of subset j are given by c_j. We define R(j) to be the set of rows that are contained in subset j:

$R(j) = \{ r \in R \mid a_{rj} = 1 \}$ [1.4]

Furthermore, we define J(r) to be the set of subsets that contain row r:

$J(r) = \{ j \in J \mid a_{rj} = 1 \}$ [1.5]

Without loss of generality, we assume that the cost vector c consists of integers. Throughout this thesis we will use the term ‘column’ when we refer to a subset of the problem and the term ‘row’ when we refer to a constraint of the problem.

1.2 Complexity

This section briefly introduces the concepts of complexity theory in relation to our research. For more information and a more formal discussion of complexity theory, see Garey and Johnson (1979) and Papadimitriou and Steiglitz (1982).

Every combinatorial optimization problem has a closely related recognition problem. The recognition version of a problem is a question that can be answered only by "yes" or "no" and is generally of the following form:

> Given an instance I and an integer L, does there exist a feasible solution with costs at most L?

Since the recognition version of an optimization problem is not harder to solve than the optimization problem itself, any negative results proved about the complexity of the recognition problem will also apply to the optimization problem (Papadimitriou and Steiglitz, 1982). The decision version of the set partitioning problem is as follows:

> Given a finite set S and a collection C = {s_1, …, s_n } of subsets of S, does C contain a collection C' of pair-wise disjoint subsets, such that $\cup_{s_i \in C'} s_i = S$?

The collection of decision problems for which a solution algorithm exists, whose complexity grows polynomially with the size of the input, is denoted by *P*. The complexity class *NP* is defined to be the class of problems for which feasibility of a solution can be checked in polynomial time. For example, the decision version of the set partitioning problem is an element of *NP*, since for every collection C', we can check in polynomial time whether $\cup_{s_i \in C'} s_i = S$. By definition, $P \subseteq NP$. It is widely believed that $P \neq NP$. The most difficult problems in the class *NP* are called *NP*-complete problems. To be able to characterize these problems, we first explain the concept of polynomial reducibility.

A decision problem Π_1 is said to be polynomially reducible to a decision problem Π_2 if, given an input I of Π_1, one can construct an input F(I) of Π_2, in time polynomial in the size of input I, such that I is a "yes"-instance for Π_1 if and only if F(I) is a "yes"-instance for Π_2.

We can now say that a decision problem Π is *NP*-complete if Π is in *NP* and every other problem in *NP* can be polynomially reduced to Π. The class of *NP*-complete problems has two important characteristics (Papadimitriou and Steiglitz, 1982):

1. There is no *NP*-complete problem for which a polynomial time solution algorithm is known.
2. If a polynomial time algorithm exists for one *NP*-complete problem, then a polynomial time algorithm exists for all *NP*-complete problems.

The well-known and widely accepted conjecture, that no polynomial time solution algorithm exists for any *NP*-complete problem, is based on these two observations. The complexity of a problem gives an indication of the difficulty of the problem and the type of solution method to use.

The set partitioning problem is an *NP*-complete problem (Karp, 1972). According to common belief, this implies that no polynomial time algorithm exists to solve set partitioning problems to optimality. However, our research aims at developing an algorithm to solve these types of problems to optimality. Due to the special structure of the problem and considering the current state of knowledge and technology, it is possible to solve to optimality large instances of the set partitioning problem in a reasonable amount of time.

1.3 Applications

Like the traveling salesman problem, the set partitioning problem is a well-studied mixed integer programming (MIP) problem. Since many real-life problems can be formulated as set partitioning problems, much research has focused on set partitioning applications. This section provides some examples of these applications.

The two most famous and successful applications of set partitioning are vehicle routing and crew scheduling. Generally, the vehicle routing problem considers a set of customers that have to be supplied by one or more vehicles. See Foster and Ryan (1976), Fleuren (1988), Borndörfer et al. (1998) and Le Blanc et al. (2004A, 2004B) for examples of vehicle routing applications. The crew scheduling problem considers a set of tasks that have to be assigned to a group of people, taking into account several constraints. For examples of applications of set partitioning to solve crew scheduling problems, see Falkner and Ryan (1987), Graves et al. (1993), Hoffman and Padberg (1993), Desaulniers et al. (1997), Mingozzi et al. (1999), Butchers et al. (2001) and Yan and Chang (2002).

Nawijn (1987) discusses an application of the set partitioning problem to optimize the performance of a blood analyzer. Baldacci et al. (2002) describe an approach to solve capacitated location problems by formulating them as a set partitioning problem. Ryan and Falkner (1988) describe how set partitioning can be used to solve scheduling problems. Cattrysse et al. (1994) present a set partitioning heuristic for solving the generalized assignment problem. Chapters 8 and 9 will describe two case study that were solved using the solver discussed in this thesis.

1.4 Literature

This section provides an overview of the literature on solving set partitioning problems. Rather than attempting to give a complete description of all literature in the area, we offer a general overview of the literature relevant for the research described in this thesis.

1.4.1 Heuristics

In many real-life situations, there is no need to have the exact optimal solution. While real-life projects often involve estimations and assumptions, one is usually satisfied with an approximation algorithm that finds a good solution quickly. Since the set partitioning problem is NP-complete, many research efforts were aimed at developing good heuristics for this problem. We will provide some examples of heuristics for set partitioning problems in the literature.

Ryan and Falkner (1988) attempt to find a good solution to the set partitioning problem by imposing additional structure to the problem that is derived from real-life applications. This method appears to be effective in finding a good feasible solution quickly. Atamtürk et al. (1995) describe a combined Lagrangian, linear programming and implication heuristic to generate provably good solutions. They also use preprocessing and probing techniques to speed up the algorithm. Their results show that the algorithm performs well in finding good, and often even optimal, solutions quickly.

The recent literature has shown much interest in evolutionary algorithms to handle hard combinatorial problems. The ideas behind evolutionary or genetic algorithms are derived from the evolutionary process of biological organisms in nature. They are based on the principles of natural selection and survival of the fittest, in such a way that the good characteristics from a pair of “ancestors” can be combined to produce even better “offspring”. An example of a genetic algorithm for the SPP can be found in Chu and Beasley (1998), who also report good results, finding optimal or near-optimal solutions very quickly for all problems in their test set.

1.4.2 Optimal solution algorithms

Although the set partitioning problem is NP-complete, there have been many research efforts to develop algorithms to solve this problem to optimality. With the current state of technology, it is possible to solve to optimality large instances of the set partitioning problem in a reasonable amount of time by making use of the special structure of the problem. This is not only due to the ongoing developments in hardware, but also in the major achievements in the development and implementation of algorithms. There are two large classes of optimization algorithms for the set partitioning problem: ‘branch and bound’ and ‘branch and cut’ algorithms.

Branch and bound algorithms are enumeration techniques to find an optimal solution, after possible pre-solving like preprocessing and bound calculations. Marsten (1974) was the first to describe a successful implementation of the branch and bound technique for set partitioning problems. This algorithm uses linear programming to calculate lower bounds to the problem. Marsten gives results for five set partitioning problems. The largest problem solved by his algorithm consists of 200 rows and 2,362 columns and was solved in less than seven minutes. The fifth and

largest problem, 419 rows and 21,585 columns, was not solved within three hours. Nevertheless, his results were highly promising at that time.

Albers (1980) describes different enumeration algorithms for the set partitioning problem. Different heuristics for lower bound determination are discussed. He reports on computational experiments on randomly generated problem instances of 20 to 70 rows and 500 to 3000 columns, most of which are solved within an hour. Ryan (1992) discusses a branch and bound algorithm for set partitioning problems that uses linear programming to find lower bounds and constraint branching to find the optimum. He reports on computing times of three hours for problems with almost 200,000 variables and 600 constraints.

Branch and cut algorithms use enumeration techniques, along with the generation of polyhedral cuts. These cuts are added to tighten the linear programming relaxation of the problem in order to improve the quality of the linear programming solution. This provides not only a better lower bound, but also valuable information to improve the branching strategy. Note that branch and cut algorithms require the use of a linear programming solver. A general discussion of valid inequalities for set partitioning problems can be found in Balas and Padberg (1976). Chapters 5 and 6 will briefly consider the use of cuts in the context of our research.

Hoffman and Padberg (1993) describe a highly successful implementation of a branch and cut solver for set partitioning problems that uses three different relaxations of the underlying polytope to generate polyhedral cuts. They discuss results on 55 set partitioning problems that are also in the test set used in this thesis, see Section 1.5. For most of these instances, the solution time is within minutes, and sometimes even seconds, which was a great improvement compared to the algorithms known at that time.

In his thesis, Borndörfer (1998) compares the branch and cut approach with a small selection of cuts to a general branch and bound approach. The difference turns out to be very small for all 55 problems in his test set, which is the same as the one used in Hoffman and Padberg (1993). Moreover, he reports on computing times in the order of seconds for almost all problems in this set. The largest computing time is slightly over five minutes for a problem with 426 rows and 7,195 columns, which took 38 hours to solve in the implementation of Hoffman and Padberg. Note that both algorithms of Borndörfer, as well as all other approaches discussed in this section, use linear programming software to determine lower bounds. An alternative method to calculate lower bounds is Lagrangian relaxation. More on Lagrangian relaxation for set partitioning problems can be found in Van Krieken et al. (2004) and in Chapter 3 of this thesis.

1.5 Test instances

Throughout this thesis, we regularly support our findings with computational results.

To this end, we have formed a test set of set partitioning problems, consisting of 60 problems. From this set, 55 instances are real-life set partitioning problems that stem from the OR-library of Beasley (Beasley, 1990). This is the same set that is used in Hoffman & Padberg (1993) and Borndörfer (1998). The other five problems are set partitioning formulations of puzzles. Three of them, *Heart*, *Meteor* and *Delta*, are parts of the well-known *Eternity* puzzle (Eternity, 2004). For a description of the *Bill's Snowflake* puzzle, see Snowflake (2004). Finally, the *Exotic Fives* puzzle is described at Exotic (2004). The last two instances will be referred to in this thesis as 'Snowflake' and 'Fives' respectively. The five new puzzle instances can also be obtained through the OR-library (OR-library, 2004). The puzzles are modeled as set partitioning problems as follows. The compartments of the puzzle are represented by the rows of the set partitioning problem. Every piece of the puzzle has several columns in the set partitioning tableau, representing the different ways that the piece can be placed in the puzzle. The constraints ensure that no more than one piece covers each compartment. Moreover, we add one constraint for every piece to ensure that this piece is used exactly once. To solve a puzzle, we just need a feasible solution to this problem. This is modeled by giving all the columns equal costs, such that we minimize the number of pieces used. This number is equal for all feasible solutions, since we have to use all the pieces.

The problem characteristics of the 60 instances are given in Table 1, where the density of a problem denotes the percentage of nonzero's in the constraint matrix. All computational experiments reported in this thesis are performed on a normal desktop computer, running on MS Windows XP with a 2.4 Ghz Pentium processor and 1536 MB RAM, unless mentioned otherwise. All algorithms are written in C.

1.6 Goal and motivation of the research

Solving set partitioning problems has been a subject of research for decades. Already in 1976, an extensive survey of set partitioning problems was published (Balas and Padberg, 1976). Since then, many efforts have been made to solve increasingly larger problems. To our knowledge, Hoffman and Padberg (1993) were the first to discuss a successful algorithm that was able to solve large problem instances to optimality. They report optimal results on real-life airline crew scheduling problems for several American airline companies. The main goal of our research is to develop a fast optimization algorithm for the set partitioning problem.

The algorithms discussed in literature that are fast and capable of solving large instances of the set partitioning problem, are all based on linear programming techniques. The use of linear programming to determine the lower bound has several advantages over other methods, such as Lagrangian relaxation. For example, information about the solution to the linear programming relaxation can be of great use in determining the branching strategy in a branch and bound algorithm.

Table 1.1: Problem characteristics

Problem	Columns	Rows	Elements	Density
nw41	197	17	740	22%
nw32	294	19	1357	24%
nw40	404	19	2069	27%
nw08	434	24	2332	22%
nw15	467	31	2830	20%
nw21	577	25	3591	25%
nw22	619	23	3399	24%
nw12	626	27	3380	20%
nw39	677	25	4494	27%
nw20	685	22	3722	25%
nw23	711	19	3350	25%
nw37	770	19	3778	26%
nw26	771	23	4215	24%
nw10	853	24	4336	21%
nw34	899	20	5045	28%
Heart	926	180	8334	5%
nw43	1072	18	4859	25%
nw42	1079	23	6533	26%
Delta	1194	126	10746	7%
nw28	1210	18	8553	39%
nw25	1217	20	7341	30%
nw38	1220	23	9071	32%
nw27	1355	22	9395	32%
nw24	1366	19	8617	33%
nw35	1709	23	10494	27%
nw36	1783	20	13160	37%
Snowflake	2300	585	103938	8%
Fives	2440	72	14640	8%
Meteor	2464	60	14784	10%
nw29	2540	18	14193	31%
nw30	2653	26	20436	30%
nw31	2662	26	19977	29%
nw19	2879	40	25193	22%
nw33	3068	23	21704	31%
nw09	3103	40	20111	16%
nw07	5172	36	41187	22%
aa02	5198	531	36359	1%
nw06	6774	50	61555	18%
aa04	7195	426	52121	2%
aa06	7292	646	51728	1%
kl01	7479	55	56242	14%
aa05	8308	801	65953	1%
aa03	8627	825	70806	1%
nw11	8820	39	57250	17%
aa01	8904	823	72965	1%
nw18	10757	124	91028	7%
us02	13635	100	192716	14%
nw13	16043	51	104541	13%
us04	28016	163	297538	7%
kl02	36699	71	212536	8%
nw03	43749	59	363939	14%
nw01	51975	135	410894	6%
us03	85552	77	1211929	18%
nw04	87482	36	636666	20%
nw02	87879	145	721736	6%
nw17	118607	61	1010039	14%
nw14	123409	73	904910	10%
nw16	148633	139	1501820	7%
nw05	288507	71	2063641	10%
us01	1053137	145	13636541	9%
Average	38585	123	405456	18%

Moreover, this information is needed to determine the value of cuts in a branch and cut method. However, the linear programming relaxation of a set partitioning problem is highly degenerate and difficult to solve (Hoffman and Padberg, 1993). Therefore, a high quality linear programming solver, which is often expensive, is needed to solve these relaxations. In the methods described in literature, CPLEX (ILOG, 2004) is often used to solve the relaxations. The goal of our research is to find out if a Lagrangean relaxation based branch and bound algorithm, without using any external mathematical programming solvers, can achieve the same kind of performance as the successful linear programming based algorithms that are described in the literature in the last decades. The most important results of this research is the set partitioning solver LaRSS: Lagrangian Relaxation Set partitioning Solver.

1.7 Outline of the thesis

The remainder of this thesis is organized as follows.

Chapter 2 deals with the concept of preprocessing. Several known and new preprocessing techniques for set partitioning are discussed and results on the test set of set partitioning problems are presented.

Chapter 3 considers lower bounding techniques. We discuss the Lagrangian relaxation of the set partitioning problem, as well as several aspects of subgradient search approaches and two dual heuristics to improve lower bounds for the set partitioning problem.

Chapter 4 discusses upper bound mechanisms. A primal heuristic to find feasible solutions is discussed and the impact of upper bounds on the branch and bound procedure is investigated by computational experiments on our set partitioning test set.

Chapter 5 deals with the branch and bound algorithm that is applied to find the optimal solution to a set partitioning problem. We discuss several branching strategies and the research that we have performed to improve the branching process.

Chapter 6 discusses several research directions that we have examined in our project. Theory and results on possible decompositions of the set partitioning tableaus are presented. Moreover, we discuss the possibility of adding cuts to improve the performance of our set partitioning solver LaRSS. Finally, we discuss some technical issues related to implementation and data management.

Chapter 7 deals with the solver LaRSSthat is developed to solve pure set partitioning problems. The composition of the solver is discussed, as well as computational results on a test set of set partitioning problems. The performance of the solver is compared to the general mixed integer solver CPLEX (ILOG, 2004).

Chapters 8 and 9 consider two real-life cases performed for Auto Recycling Nederland. In these cases our solver is used to solve large series of vehicle routing

problems.

Chapter 10 discusses the extension of the problem space of the solver to the more general set partitioning problem with side-constraints. Again we compare the performance of the solver to CPLEX.

Finally, Chapter 11, summarizes our experiences with the methodology presented in this thesis, discusses its strengths and weaknesses, and proposes topics for further research.

Chapter 2

Preprocessing

This chapter describes preprocessing techniques that are designed to reduce the solution time of set partitioning problems. These techniques preserve the set partitioning formulation and therefore can be applied in any solution algorithm for set partitioning problems. Besides a brief review of the existing literature on preprocessing set partitioning problems, we also present several new techniques. The different preprocessing techniques are discussed in Sections 2.2 and 2.3. Section 2.5 establishes several relationships between the techniques. The value of the techniques is illustrated by various computational experiments, discussed in Sections 2.4 and 2.6. Finally, Section 2.7 summarizes our findings.

2.1 Introduction

Preprocessing is a generic term for all techniques designed to improve the formulation of linear or integer programs, such that they can be solved more rapidly by some solution method. Mostly, these techniques use logical implications to simplify a problem in an automated way. Probing techniques investigate the consequences of tentatively setting a binary variable to 0 or 1. More on preprocessing and probing techniques for general mixed integer programming problems can be found in Savelsbergh (1994). This chapter focuses on preprocessing techniques developed especially for set partitioning problems. These techniques aim to reduce the number of columns and/or the number of rows of the problem in order to reduce the total time needed to solve the problem.

For all tables that are given in this chapter, we use the following notation:

- CR: column reduction
- %CR: percentage column reduction
- RR: row reduction

- %RR: percentage row reduction
- T: time in seconds

2.2 Preprocessing rules for the set partitioning problem

This section discusses several pure reduction techniques. Results considering these techniques are discussed in Section 2.4 and relationships between the different preprocessing techniques are examined in Section 2.5. The implementation of the techniques is described in Appendix A.

2.2.1 Equal k-columns

If a column j can be represented by a combination of k other columns, k > 0, with less costs, then column j can be removed from the problem. More formally:
If $R(j) = \cup_{i \in K} R(i)$, $R(i_1) \cap R(i_2) = \emptyset \ \forall i_1, i_2 \in K$ and $c(j) \geq \sum_{i \in K} c(i)$ for $j \in J, K \subset J \setminus j$, then column j can be removed from the problem.
The well-known equal columns preprocessing rule (see for example Hoffman and Padberg, 1993) is a special case of this rule, with k = 1. Although the concept of this preprocessing rule is very straightforward, equal columns occur frequently, since many real-life set partitioning problems are constructed by explicit heuristic generation techniques.

2.2.2 Equal rows

If two rows are covered by the same set of columns, one of these rows can be removed from the problem. More formally:
If $J(r) = J(s)$ for $r, s \in R$, then row r can be removed from the problem.

Equal rows, or identical constraints, are not likely to occur in a real-life set partitioning problem. However, equal rows can result from applying other preprocessing techniques. The computational experiments described in the next section illustrate how this simple and quick rule can complement other preprocessing techniques.

2.2.3 k-Rowsets and contained rows

If there is a set of k rows, $r_1, \ldots, r_k$, k > 1, for which it holds that there is no column j for which $r_1 \in R(j)$ and $r_2, \ldots, r_k \notin R(j)$, then all columns c for which $r_1 \notin R(c)$ and $r_2, \ldots, r_k \in R(c)$ can be deleted.
In the case k = 2, the resulting rows r_1 and r_2 are equal and the equal rows rule can be applied to delete one of them. This is equivalent to applying the well-known

contained rows preprocessing rule, which states that if row r is contained in another row s, then all columns that cover row s, but not row r, as well as row s, can be removed from the problem. More formally:
If $J(r) \subseteq J(s)$ for $r,s \in R$, then all $j \in J(s)/J(r)$ and s can be removed from the problem.

The contained rows preprocessing rule can also be found in the literature on set partitioning problems, see for example Hoffman and Padberg (1993). This preprocessing rule is particularly interesting, since it can lead to a reduction in columns as well as rows. The 3-rowset rule is also discussed in the literature, see for example Borndörfer (1998), who refers to this rule as the symmetric difference rule. When k increases, finding k-rowsets becomes more time-consuming. Section 2.3.2 considers computational results of the contained rows rule as well as the 3-rowset rule.

2.2.4 Clique rule

If all columns that cover row r have one or more elements in common with a column j that does not cover row r, then we can remove column j, since choosing this column in a solution set will leave constraint r unsatisfiable. Another way to formulate this is as follows (Hoffman & Padberg, 1993). Derive a graph from the set partitioning problem where the nodes of the graph correspond to the columns and two nodes are connected if they share at least one element. A trivial clique C_r in such a graph is the set of all nodes (columns) containing a certain element r of the ground set R. This implies that every feasible solution contains only one element of this clique. If we can find a clique C that properly subsumes C_r, then every column in $C \backslash C_r$ can be removed.

2.2.5 Cut rule

For a given set of three rows {r,s,t}, we define CS(r,s,t) as the set of columns that cover at least two of rows r, s and t:

$$CS(r,s,t) = \{j \in J \mid |R(j) \cap \{r,s,t\}| \geq 2\} \qquad [2.1]$$

The cut rule says that if we can find a row w for which $J(w) \subseteq CS(r,s,t)$, then we can delete all columns in the set $CS(r,s,t) - J(w)$. More intuitively, this can be explained as follows. For the set of rows {r,s,t}, we can discern four types of subsets:

$$T_n(r,s,t) = \{j \in J \mid |R(j) \cap \{r,s,t\}| = n\} \quad n = 0,1,2,3 \qquad [2.2]$$

Thus, $T_n(r,s,t)$ denotes the set of columns that cover n of the rows {r,s,t}.

A solution to the set partitioning problem contains at most one of all the columns that are incorporated in $T_2(r,s,t)$ and $T_3(r,s,t)$. This actually forms a cut to the set partitioning problem, which covers all columns $j \in T_2(r,s,t) \cup T_3(r,s,t)$. If we can find a row w that is contained in this cut, we can delete all columns that are in the cut,

but not in J(w). The cut rule is developed by the authors and first described in Van Krieken et al. (2003).

2.3 Row combination technique

This section discusses a new technique designed to reduce the number of constraints in the problem. To this end, a small increase in the number of columns can be allowed. Below, we will discuss the technique and the implementation. Computational results considering this technique are discussed in the next section.

2.3.1 Technique

When we say that we combine two rows r_1 and r_2, we mean the following:
For every column $j_1 \in J(r_1)$, $j_1 \notin J(r_2)$
For every column $j_2 \in J(r_2)$, $j_2 \notin J(r_1)$
    If $R(j_1) \cap R(j_2) = \varnothing$
        Make a new column j_3 for which $R(j_3) = R(j_1) \cup R(j_2)$ and $c_{j_3} = c_{j_1} + c_{j_2}$
Delete all columns $j_1 \in J(r_1)$, $j_1 \notin J(r_2)$ and $j_2 \in J(r_2)$, $j_2 \notin J(r_1)$
Delete row r1 or row r2 arbitrarily

Combining rows r_1 and r_2 thus means that we add all combinations of columns that cover only one of the two rows. Since we add all combinations, the columns that cover only one of the two rows can be deleted from the problem. After this step, rows r_1 and r_2 are equal, so we can delete one of the rows. This makes the technique particularly interesting for pairs of rows that differ only on a few elements, since in that case we only add a few columns, while we can remove one row. It can even be the case both the number of rows and the number of columns of the problem decreases. When rows are combined, these combinations must be memorized in such a way that when a solution to the problem is found, the original columns of which this solution consists can be reconstructed.

2.3.2 Implementation

The performance of the row combination technique obviously depends on how the pairs of rows are selected. We implemented the technique as follows:

Step 0: Max_growth = $\frac{p}{100} \cdot$ number of columns.

Step 1: For each $r_1, r_2 \in R$ we define:

$$C(r_1, r_2) = \{j \in J(r_1) \mid j \notin J(r_2)\} \qquad [2.3]$$

and

$$f(r_1, r_2) = |C(r_1, r_2)| \cdot |C(r_2, r_1)| - |C(r_1, r_2)| - |C(r_2, r_1)| \qquad [2.4]$$

This function gives an upperbound on the increase in the number of columns when rows r_1 and r_2 are combined. Now, let {s,t} be the set of rows for which $f(r_1, r_2)$ is minimal.
If (f(s, t) > Max_growth) then stop.

Step 2: Combine rows s and t. Now delete all columns $k \in \{j \in J(s) \mid j \notin J(t)\}$ and all columns $m \in \{j \in J(t) \mid j \notin J(s)\}$ and row s. Go to step 1.

This implementation uses the parameter p, a percentage that denotes the maximal allowed growth in the number of columns. Extensive testing should be used to determine the optimal value of this parameter. In our experience, the technique works well with small values of p, typically between 0 and 2. Since the value of f(s,t) is an upper bound on the increase in the number columns when rows s and t are combined, the actual increase in columns will generally be smaller. Furthermore, as will be shown by the computational results in the next section, the number of rows can be reduced significantly if we allow a small increase in the number of columns. Besides, experience shows that, for typical set partitioning problems, the number of rows has a greater influence on the computing time of a solution than the number of columns. These observations illustrate that it can be effective to take a small but positive value for the parameter p.

2.3.3 Row combination technique as preprocessing rule

The row combination technique serves as a pure problem reduction rule when the parameter p is given the value 0. In this case, two rows will be combined only if the number of columns does not increase. Furthermore, when the rows are combined, one of them will be deleted. As the computational results in the next section will show, the reductions achieved with p = 0 are considerable.

2.4 Individual computational results

This section discusses computational results for the rules discussed in Sections 2.2 and 2.3. Computational experiments for different sequences of preprocessing techniques are discussed in Sections 2.4 and 2.5. The implementation of the preprocessing techniques is discussed in Appendix A.

Table 2.1: Results of the equal columns and equal 2-columns preprocessing rules

			Equal columns			Equal 2-columns			Equal k-columns		
Name	**Cols**	**Rows**	**CR**	**%CR**	**T**	**CR**	**%CR**	**T**	**CR**	**%CR**	**T**
nw41	197	17	20	10%	0.00	109	55%	0.00	113	57%	0.00
nw32	294	19	42	14%	0.00	150	51%	0.00	168	57%	0.00
nw40	404	19	68	17%	0.00	176	44%	0.00	179	44%	0.00
nw08	434	24	78	18%	0.00	243	56%	0.00	350	81%	0.00
nw15	467	31	2	0%	0.00	8	2%	0.00	8	2%	0.00
nw21	577	25	151	26%	0.00	333	58%	0.02	375	65%	0.02
nw22	619	23	88	14%	0.00	262	42%	0.00	278	45%	0.02
nw12	626	27	172	27%	0.00	303	48%	0.02	525	84%	0.00
nw39	677	25	110	16%	0.00	347	51%	0.00	431	64%	0.02
nw20	685	22	119	17%	0.00	299	44%	0.00	321	47%	0.02
nw23	711	19	237	33%	0.00	428	60%	0.00	438	62%	0.02
nw37	770	19	131	17%	0.02	421	55%	0.00	477	62%	0.02
nw26	771	23	229	30%	0.00	412	53%	0.02	446	58%	0.02
nw10	853	24	194	23%	0.00	504	59%	0.00	769	90%	0.02
nw34	899	20	149	17%	0.00	469	52%	0.00	502	56%	0.03
Heart	926	180	26	3%	0.00	26	3%	0.00	26	3%	0.06
nw43	1072	18	89	8%	0.00	528	49%	0.02	539	50%	0.05
nw42	1079	23	184	17%	0.00	397	37%	0.00	431	40%	0.03
Delta	1194	126	52	4%	0.00	52	4%	0.00	52	4%	0.09
nw28	1210	18	385	32%	0.00	600	50%	0.02	612	51%	0.03
nw25	1217	20	373	31%	0.00	861	71%	0.02	919	76%	0.03
nw38	1220	23	309	25%	0.00	444	36%	0.02	444	36%	0.03
nw27	1355	22	429	32%	0.00	805	59%	0.02	892	66%	0.03
nw24	1366	19	440	32%	0.00	946	69%	0.00	1168	86%	0.03
nw35	1709	23	306	18%	0.00	958	56%	0.02	1022	60%	0.09
nw36	1783	20	375	21%	0.00	484	27%	0.02	485	27%	0.08
Snowflake	2300	585	0	0%	0.00	0	0%	0.03	0	0%	1.78
Fives	2440	72	0	0%	0.00	0	0%	0.03	0	0%	0.27
Meteor	2464	60	774	31%	0.00	774	31%	0.03	774	31%	0.13
nw29	2540	18	506	20%	0.00	1005	40%	0.06	1013	40%	0.16
nw30	2653	26	769	29%	0.00	1756	66%	0.03	1984	75%	0.17
nw31	2662	26	839	32%	0.00	1628	61%	0.05	1735	65%	0.19
nw19	2879	40	734	25%	0.00	1545	54%	0.05	1617	56%	0.31
nw33	3068	23	653	21%	0.00	1524	50%	0.06	1591	52%	0.28
nw09	3103	40	798	26%	0.00	1972	64%	0.06	2379	77%	0.23
nw07	5172	36	2064	40%	0.00	3567	69%	0.11	3892	75%	0.53
aa02	5198	531	0	0%	0.00	160	3%	0.20	161	3%	1.61
nw06	6774	50	797	12%	0.02	2095	31%	0.39	2282	34%	2.22
aa04	7195	426	0	0%	0.00	197	3%	0.39	200	3%	3.97
aa06	7292	646	0	0%	0.00	384	5%	0.41	392	5%	3.88
kl01	7479	55	676	9%	0.00	676	9%	0.48	676	9%	2.19
aa05	8308	801	0	0%	0.02	264	3%	0.53	270	3%	6.14
aa03	8627	825	0	0%	0.02	342	4%	0.58	347	4%	6.75
nw11	8820	39	2332	26%	0.00	5775	65%	0.41	8217	93%	1.48
aa01	8904	823	0	0%	0.00	196	2%	0.66	196	2%	7.19
nw18	10757	124	2297	21%	0.00	4136	38%	0.94	4169	39%	4.86
us02	13635	100	2256	17%	0.02	2594	19%	1.36	2642	19%	10.92
nw13	16043	51	5138	32%	0.02	9910	62%	1.30	14742	92%	4.31
us04	28016	163	13001	46%	0.02	17828	64%	3.81	18077	65%	20.19
kl02	36699	71	20157	55%	0.06	20157	55%	6.59	20157	55%	20.53
nw03	43749	59	4785	11%	0.06	14708	34%	20.94	15363	35%	152.06
nw01	51975	135	1906	4%	0.14	16711	32%	30.16	27080	52%	256.24
us03	85552	77	39362	46%	0.45	57824	68%	50.88	60406	71%	307.80
nw04	87482	36	41292	47%	0.17	41313	47%	54.05	41313	47%	177.08
nw02	87879	145	2621	3%	0.17	28161	32%	84.95	46357	53%	2168.00
nw17	118607	61	40421	34%	0.33	78070	66%	105.88	96721	82%	1265.09
nw14	123409	73	28231	23%	0.33	68694	56%	130.56	117246	95%	1329.08
nw16	148633	139	9682	7%	0.34	9682	7%	199.28	9682	7%	7142.00
nw05	288507	71	85904	30%	0.89	193578	67%	410.86	260174	90%	8497.00
us01	1053137	145	682495	65%	3.81	829147	79%	5622.00	853434	81%	68498.00
Average	38585	123	16587	20%	0.11	23782	41%	112.14	27121	48%	1498.22

Table 2.2: Results of the contained rows and 3-rowset preprocessing rules

			Contained rows					3-rowset + equal rows				
Name	**Cols**	**Rows**	**CR**	**RR**	**%CR**	**%RR**	**T**	**CR**	**RR**	**%CR**	**%RR**	**T**
nw41	197	17	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw32	294	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw40	404	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw08	434	24	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw15	467	31	0	0	0%	0%	0.00	58	0	12%	0%	0.00
nw21	577	25	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw22	619	23	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw12	626	27	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw39	677	25	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw20	685	22	0	0	0%	0%	0.00	28	0	4%	0%	0.00
nw23	711	19	0	0	0%	0%	0.00	53	0	7%	0%	0.00
nw37	770	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw26	771	23	0	0	0%	0%	0.00	122	2	16%	9%	0.00
nw10	853	24	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw34	899	20	0	0	0%	0%	0.00	0	0	0%	0%	0.00
Heart	926	180	0	44	0%	24%	0.02	36	44	4%	24%	0.00
nw43	1072	18	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw42	1079	23	0	0	0%	0%	0.00	51	0	5%	0%	0.00
Delta	1194	126	0	10	0%	8%	0.00	94	10	8%	8%	0.00
nw28	1210	18	0	0	0%	0%	0.00	315	0	26%	0%	0.00
nw25	1217	20	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw38	1220	23	0	0	0%	0%	0.00	119	0	10%	0%	0.00
nw27	1355	22	0	0	0%	0%	0.00	91	0	7%	0%	0.02
nw24	1366	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw35	1709	23	0	0	0%	0%	0.00	126	0	7%	0%	0.00
nw36	1783	20	0	0	0%	0%	0.00	214	0	12%	0%	0.02
Snowflake	2300	585	265	15	12%	3%	1.61	270	15	12%	3%	0.02
Fives	2440	72	0	0	0%	0%	0.00	0	0	0%	0%	0.00
Meteor	2464	60	0	0	0%	0%	0.02	48	0	2%	0%	0.00
nw29	2540	18	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw30	2653	26	0	0	0%	0%	0.00	0	0	0%	0%	0.02
nw31	2662	26	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw19	2879	40	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw33	3068	23	0	0	0%	0%	0.00	62	0	2%	0%	0.00
nw09	3103	40	0	0	0%	0%	0.02	0	0	0%	0%	0.00
nw07	5172	36	0	0	0%	0%	0.02	0	0	0%	0%	0.02
aa02	5198	531	959	135	18%	25%	0.06	1069	142	21%	27%	0.02
nw06	6774	50	0	0	0%	0%	0.00	0	0	0%	0%	0.02
aa04	7195	426	752	76	10%	18%	0.06	820	77	11%	18%	0.02
aa06	7292	646	1081	124	15%	19%	0.11	1250	132	17%	20%	0.03
kl01	7479	55	863	8	12%	15%	0.02	905	8	12%	15%	0.02
aa05	8308	801	1516	219	18%	27%	0.20	1699	234	20%	29%	0.05
aa03	8627	825	1400	233	16%	28%	0.22	1684	249	20%	30%	0.05
nw11	8820	39	0	0	0%	0%	0.00	0	0	0%	0%	0.00
aa01	8904	823	1152	180	13%	22%	0.23	1252	190	14%	23%	0.03
nw18	10757	124	0	0	0%	0%	0.03	1	0	0%	0%	0.02
us02	13635	100	2844	55	21%	55%	0.06	5726	55	42%	55%	0.09
nw13	16043	51	0	0	0%	0%	0.02	0	0	0%	0%	0.02
us04	28016	163	14428	49	51%	30%	0.17	15951	50	57%	31%	0.13
kl02	36699	71	0	2	0%	3%	0.05	0	2	0%	3%	0.05
nw03	43749	59	0	0	0%	0%	0.06	0	0	0%	0%	0.06
nw01	51975	135	0	0	0%	0%	0.17	168	0	0%	0%	0.09
us03	85552	77	30368	25	35%	32%	0.45	32117	25	38%	32%	0.58
nw04	87482	36	0	0	0%	0%	0.08	0	0	0%	0%	0.13
nw02	87879	145	0	0	0%	0%	0.39	2	0	0%	0%	0.17
nw17	118607	61	0	0	0%	0%	0.31	0	0	0%	0%	0.20
nw14	123409	73	0	0	0%	0%	0.31	0	0	0%	0%	0.22
nw16	148633	139	0	0	0%	0%	1.42	0	0	0%	0%	0.27
nw05	288507	71	0	0	0%	0%	0.89	0	0	0%	0%	0.61
us01	1053137	145	53729	59	5%	41%	15.27	70244	59	7%	41%	17.19
Average	38585	123	1823	21	4%	6%	0.37	2243	22	7%	6%	0.34

2.4.1 Equal k-columns

Table 2.1 provides the results of applying the equal columns as well as the equal 2-columns and equal k-columns rules on our test set. Applying the equal columns rule on our test set reduces the number of columns for almost all instances. The largest gain is for problem us01, where the number of columns is reduced by 65% in less than four seconds. This rule is very fast and very effective in reducing the number of columns. This phenomenon is caused by the fact that many real-life set partitioning problems are constructed by generating a lot of combinations, e.g. routes or crew pairings, in an automated way, creating doubles.

The equal 2-columns rule and equal k-columns rules achieve very large column reductions; the computing times are too large, however, for these preprocessing techniques to be useful in a solution algorithm. The k-columns rule achieves an average column reduction that is over 3,000 columns higher than the 2-columns rule; the computing time of the k-columns rule, however, is 13 times as high as the time of the 2-columns rule.

2.4.2 k-Rowsets and contained rows

Table 2.2 shows the results of applying the contained rows rule and of applying the k-rowsets rule and equal rows rules consecutively. The contained rows preprocessing rule is not effective in all instances, although the reductions found are substantial, while the computing time is relatively small for all instances. The reductions found by applying the 3-rowset and equal rows rules, as well as the computation time needed, are somewhat higher than those of the contained rows rule. Section 2.5 examines the value of these rules in a preprocessing sequence.

2.4.3 Clique and equal rows

Applying the clique rule can result in equal rows. Therefore, Table 2.3 shows the results of applying the clique and equal rows rules consecutively on our test set. As can be seen, the reductions found are considerable, with column reductions up to 58% and row reductions up to 55%, while computing times are quite long for some instances. Still, as will be shown in Sections 2.5 and 2.6, the clique rule can be very effective in a set partitioning solution algorithm.

2.4.4 Cut and equal rows

After applying the cut rule, equal rows can occur. Table 2.3 shows the results of applying the cut and equal rows rules consecutively on the problems in our test set. The reductions are considerable, with column reductions up to 26% and row reductions up to 54%, although the computing time is long in some instances.

Section 2.3 examines the value of this technique in a preprocessing sequence.

Table 2.3: Results of the clique and cut preprocessing rules

			Clique + equal rows					Cut + equal rows				
Name	**Cols**	**Rows**	**CR**	**RR**	**%CR**	**%RR**	**T**	**CR**	**RR**	**%CR**	**%RR**	**T**
nw41	197	17	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw32	294	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw40	404	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw08	434	24	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw15	467	31	58	0	12%	0%	0.00	0	0	0%	0%	0.00
nw21	577	25	5	0	1%	0%	0.00	0	0	0%	0%	0.00
nw22	619	23	12	0	2%	0%	0.00	0	0	0%	0%	0.00
nw12	626	27	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw39	677	25	2	0	0%	0%	0.00	0	0	0%	0%	0.00
nw20	685	22	30	0	4%	0%	0.00	0	0	0%	0%	0.00
nw23	711	19	93	0	13%	0%	0.00	0	0	0%	0%	0.00
nw37	770	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw26	771	23	123	2	16%	9%	0.00	12	2	2%	9%	0.00
nw10	853	24	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw34	899	20	39	0	4%	0%	0.00	0	0	0%	0%	0.00
Heart	926	180	42	44	5%	24%	0.02	10	44	1%	24%	0.00
nw43	1072	18	0	0	0%	0%	0.02	0	0	0%	0%	0.00
nw42	1079	23	118	0	11%	0%	0.00	0	0	0%	0%	0.00
Delta	1194	126	143	10	12%	8%	0.02	26	10	2%	8%	0.00
nw28	1210	18	384	0	32%	0%	0.00	0	0	0%	0%	0.00
nw25	1217	20	0	0	0%	0%	0.00	0	0	0%	0%	0.02
nw38	1220	23	232	0	19%	0%	0.00	0	0	0%	0%	0.02
nw27	1355	22	137	0	10%	0%	0.00	0	0	0%	0%	0.00
nw24	1366	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw35	1709	23	256	0	15%	0%	0.02	0	0	0%	0%	0.00
nw36	1783	20	265	0	15%	0%	0.00	0	0	0%	0%	0.02
Snowflake	2300	585	280	12	12%	2%	0.05	0	0	0%	0%	0.08
Fives	2440	72	0	0	0%	0%	0.02	0	0	0%	0%	0.00
Meteor	2464	60	437	0	18%	0%	0.02	0	0	0%	0%	0.00
nw29	2540	18	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw30	2653	26	10	0	0%	0%	0.00	0	0	0%	0%	0.00
nw31	2662	26	125	0	5%	0%	0.00	0	0	0%	0%	0.00
nw19	2879	40	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw33	3068	23	112	0	4%	0%	0.02	0	0	0%	0%	0.00
nw09	3103	40	0	0	0%	0%	0.02	0	0	0%	0%	0.00
nw07	5172	36	0	0	0%	0%	0.02	0	0	0%	0%	0.02
aa02	5198	531	1169	136	22%	26%	0.17	345	68	7%	13%	0.22
nw06	6774	50	0	0	0%	0%	0.02	0	0	0%	0%	0.00
aa04	7195	426	930	73	13%	17%	0.20	258	29	4%	7%	0.11
aa06	7292	646	1145	119	16%	18%	0.33	257	38	4%	6%	0.23
kl01	7479	55	905	8	12%	15%	0.06	142	3	2%	5%	0.02
aa05	8308	801	1697	216	20%	27%	0.52	445	70	5%	9%	0.48
aa03	8627	825	1696	235	20%	28%	0.50	335	92	4%	11%	0.92
nw11	8820	39	0	0	0%	0%	0.02	0	0	0%	0%	0.00
aa01	8904	823	1231	182	14%	22%	0.55	229	56	3%	7%	0.64
nw18	10757	124	2	0	0%	0%	0.06	0	0	0%	0%	0.00
us02	13635	100	6194	55	45%	55%	0.30	1509	54	11%	54%	1.44
nw13	16043	51	0	0	0%	0%	0.03	0	0	0%	0%	0.00
us04	28016	163	16250	46	58%	28%	0.72	7182	31	26%	19%	1.11
kl02	36699	71	0	2	0%	3%	0.38	0	2	0%	3%	0.02
nw03	43749	59	0	0	0%	0%	0.16	0	0	0%	0%	0.00
nw01	51975	135	168	0	0%	0%	0.84	0	0	0%	0%	0.00
us03	85552	77	38458	23	45%	30%	10.59	6736	16	8%	21%	1.22
nw04	87482	36	0	0	0%	0%	0.33	0	0	0%	0%	0.02
nw02	87879	145	2	0	0%	0%	2.20	0	0	0%	0%	0.00
nw17	118607	61	0	0	0%	0%	0.59	0	0	0%	0%	0.02
nw14	123409	73	0	0	0%	0%	0.78	0	0	0%	0%	0.03
nw16	148633	139	0	0	0%	0%	2.69	0	0	0%	0%	0.02
nw05	288507	71	0	0	0%	0%	1.88	0	0	0%	0%	0.08
us01	1053137	145	86197	59	8%	41%	194.41	11902	58	1%	40%	332.70
Average	38585	123	2649	20	8%	6%	3.64	490	10	1%	4%	5.66

Table 2.4: Results of the row combination technique (p = 0.0, p = 0.5)

			p = 0.0					p = 0.5				
Name	**Cols**	**Rows**	**CR**	**RR**	**%CR**	**%RR**	**T**	**CR**	**RR**	**%CR**	**%RR**	**T**
nw41	197	17	0	0	0%	0%	0.00	1	2	1%	12%	0.00
nw32	294	19	0	0	0%	0%	0.00	2	3	1%	16%	0.00
nw40	404	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw08	434	24	0	0	0%	0%	0.00	5	5	1%	21%	0.00
nw15	467	31	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw21	577	25	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw22	619	23	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw12	626	27	0	0	0%	0%	0.00	-2	3	0%	11%	0.00
nw39	677	25	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw20	685	22	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw23	711	19	0	0	0%	0%	0.00	1	1	0%	5%	0.00
nw37	770	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw26	771	23	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw10	853	24	0	0	0%	0%	0.00	4	4	0%	17%	0.00
nw34	899	20	0	0	0%	0%	0.00	0	0	0%	0%	0.00
Heart	926	180	0	44	0%	24%	0.03	0	44	0%	24%	0.05
nw43	1072	18	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw42	1079	23	0	0	0%	0%	0.00	2	2	0%	9%	0.00
Delta	1194	126	0	10	0%	8%	0.02	33	12	3%	10%	0.02
nw28	1210	18	0	0	0%	0%	0.00	126	1	10%	6%	0.00
nw25	1217	20	0	0	0%	0%	0.02	0	0	0%	0%	0.00
nw38	1220	23	0	0	0%	0%	0.02	2	2	0%	9%	0.00
nw27	1355	22	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw24	1366	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw35	1709	23	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw36	1783	20	0	0	0%	0%	0.00	0	0	0%	0%	0.00
Snowflake	2300	585	265	15	12%	3%	0.45	292	415	13%	71%	8.81
Fives	2440	72	0	0	0%	0%	0.00	0	0	0%	0%	0.00
Meteor	2464	60	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw29	2540	18	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw30	2653	26	0	0	0%	0%	0.00	0	0	0%	0%	0.02
nw31	2662	26	0	0	0%	0%	0.02	0	0	0%	0%	0.00
nw19	2879	40	0	0	0%	0%	0.00	8	8	0%	20%	0.02
nw33	3068	23	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw09	3103	40	0	0	0%	0%	0.00	7	7	0%	18%	0.02
nw07	5172	36	0	0	0%	0%	0.02	3	3	0%	8%	0.02
aa02	5198	531	1242	165	24%	31%	0.50	1008	235	19%	44%	0.80
nw06	6774	50	0	0	0%	0%	0.00	12	12	0%	24%	0.05
aa04	7195	426	995	83	14%	19%	0.44	801	132	11%	31%	0.73
aa06	7292	646	1139	135	16%	21%	0.73	1016	255	14%	39%	1.52
kl01	7479	55	863	8	12%	15%	0.05	863	8	12%	15%	0.05
aa05	8308	801	1877	256	23%	32%	1.56	1638	406	20%	51%	2.64
aa03	8627	825	1577	259	18%	31%	1.78	1453	413	17%	50%	2.88
nw11	8820	39	0	0	0%	0%	0.00	10	10	0%	26%	0.13
aa01	8904	823	1217	203	14%	25%	1.55	967	330	11%	40%	2.66
nw18	10757	124	0	0	0%	0%	0.02	35	47	0%	38%	0.66
us02	13635	100	2844	55	21%	55%	0.72	3605	57	26%	57%	1.00
nw13	16043	51	0	0	0%	0%	0.02	2	2	0%	4%	0.06
us04	28016	163	18576	62	66%	38%	1.44	20409	93	73%	57%	2.36
kl02	36699	71	0	2	0%	3%	0.17	0	2	0%	3%	0.16
nw03	43749	59	0	0	0%	0%	0.13	6	6	0%	10%	0.42
nw01	51975	135	0	0	0%	0%	0.16	-48	3	0%	2%	0.58
us03	85552	77	36780	27	43%	35%	5.28	36780	27	43%	35%	5.13
nw04	87482	36	0	0	0%	0%	0.27	1	1	0%	3%	0.39
nw02	87879	145	0	0	0%	0%	0.30	-226	2	0%	1%	0.84
nw17	118607	61	0	0	0%	0%	0.44	7	7	0%	11%	1.31
nw14	123409	73	0	0	0%	0%	0.41	4	4	0%	5%	1.38
nw16	148633	139	0	0	0%	0%	0.66	7	7	0%	5%	1.98
nw05	288507	71	0	0	0%	0%	1.13	10	10	0%	14%	4.66
us01	1053137	145	53729	59	5%	41%	156.59	53729	59	5%	41%	156.09
Average	38585	123	2018	23	4%	6%	2.91	2043	44	5%	14%	3.29

Table 2.5: Results of the row combination technique (p = 1.0, p = 2.0)

			p = 1.0					p = 2.0				
Name	**Cols**	**Rows**	**CR**	**RR**	**%CR**	**%RR**	**T**	**CR**	**RR**	**%CR**	**%RR**	**T**
nw41	197	17	1	2	1%	12%	0.00	1	2	1%	12%	0.00
nw32	294	19	2	3	1%	16%	0.00	2	3	1%	16%	0.00
nw40	404	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw08	434	24	5	5	1%	21%	0.00	5	5	1%	21%	0.00
nw15	467	31	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw21	577	25	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw22	619	23	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw12	626	27	-2	3	0%	11%	0.02	-6	4	-1%	15%	0.00
nw39	677	25	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw20	685	22	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw23	711	19	-4	2	-1%	11%	0.00	-4	2	-1%	11%	0.00
nw37	770	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw26	771	23	7	2	1%	9%	0.02	7	2	1%	9%	0.00
nw10	853	24	4	4	0%	17%	0.00	4	4	0%	17%	0.00
nw34	899	20	0	0	0%	0%	0.00	0	0	0%	0%	0.00
Heart	926	180	0	44	0%	24%	0.06	-16	57	-2%	32%	0.06
nw43	1072	18	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw42	1079	23	2	2	0%	9%	0.00	2	2	0%	9%	0.00
Delta	1194	126	33	12	3%	10%	0.02	42	14	4%	11%	0.02
nw28	1210	18	126	1	10%	6%	0.00	126	1	10%	6%	0.00
nw25	1217	20	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw38	1220	23	2	2	0%	9%	0.00	2	2	0%	9%	0.00
nw27	1355	22	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw24	1366	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw35	1709	23	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw36	1783	20	0	0	0%	0%	0.00	0	0	0%	0%	0.00
Snowflake	2300	585	-204	468	-9%	80%	10.16	-228	469	-10%	80%	10.13
Fives	2440	72	0	0	0%	0%	0.00	0	0	0%	0%	0.00
Meteor	2464	60	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw29	2540	18	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw30	2653	26	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw31	2662	26	0	0	0%	0%	0.00	0	0	0%	0%	0.02
nw19	2879	40	8	8	0%	20%	0.02	8	8	0%	20%	0.02
nw33	3068	23	0	0	0%	0%	0.00	0	0	0%	0%	0.02
nw09	3103	40	7	7	0%	18%	0.02	-11	9	0%	23%	0.02
nw07	5172	36	3	3	0%	8%	0.02	3	3	0%	8%	0.02
aa02	5198	531	807	241	16%	45%	0.84	61	257	1%	48%	0.97
nw06	6774	50	12	12	0%	24%	0.05	12	12	0%	24%	0.06
aa04	7195	426	491	138	7%	32%	0.78	-585	150	-8%	35%	0.89
aa06	7292	646	528	269	7%	42%	1.66	-592	284	-8%	44%	1.88
kl01	7479	55	863	8	12%	15%	0.05	863	8	12%	15%	0.06
aa05	8308	801	756	432	9%	54%	2.95	-341	450	-4%	56%	3.28
aa03	8627	825	924	428	11%	52%	3.06	-377	444	-4%	54%	3.36
nw11	8820	39	11	11	0%	28%	0.19	11	11	0%	28%	0.19
aa01	8904	823	403	343	5%	42%	2.86	-626	357	-7%	43%	3.11
nw18	10757	124	-49	49	0%	40%	0.74	-49	49	0%	40%	0.73
us02	13635	100	3605	57	26%	57%	1.00	3605	57	26%	57%	1.02
nw13	16043	51	2	2	0%	4%	0.06	2	2	0%	4%	0.06
us04	28016	163	20409	93	73%	57%	2.36	20145	94	72%	58%	2.39
kl02	36699	71	0	2	0%	3%	0.17	0	2	0%	3%	0.17
nw03	43749	59	6	6	0%	10%	0.42	6	6	0%	10%	0.42
nw01	51975	135	-48	3	0%	2%	0.58	-48	3	0%	2%	0.58
us03	85552	77	36780	27	43%	35%	5.13	36173	30	42%	39%	6.22
nw04	87482	36	1	1	0%	3%	0.41	1	1	0%	3%	0.34
nw02	87879	145	-226	2	0%	1%	0.84	-226	2	0%	1%	0.84
nw17	118607	61	7	7	0%	11%	1.28	7	7	0%	11%	1.28
nw14	123409	73	5	5	0%	7%	11.42	5	5	0%	7%	11.39
nw16	148633	139	8	8	0%	6%	3.61	8	8	0%	6%	3.55
nw05	288507	71	10	10	0%	14%	3.95	10	10	0%	14%	3.95
us01	1053137	145	53729	59	5%	41%	162.06	53729	59	5%	41%	162.13
Average	38585	123	1984	46	4%	15%	3.61	1862	48	2%	16%	3.65

2.4.5 Row combination technique

Table 2.4 shows the results of the row combination technique on our test set for p = 0 and p = 0.5. Results for p = 1.0 and p = 2.0 are given in Table 2.5. For all values of p, the computing time grows with the size of the reductions found. This is caused by the efforts made to add columns and administrate the changes.

When p increases, both the total computing time and the number of deleted rows grows. However, the amount of deleted columns decreases and even becomes negative for some instances. Moreover, the number of added columns grows rapidly compared to the reduction in the number of rows. For example, for problem aa02, the amount of deleted columns go from 1,242 when p = 0 to 807 when p = 1 and 61 when p = 2. The reduction in the number of rows goes from 165 to 241 to 257. This observation indicates that the technique works best for small values of p. In LaRSS, we use the row combination technique with p = 0.5.

2.5 Links between the different techniques

2.5.1 Relationship between contained rows and clique techniques

Theorem 2.1 All reductions that are found by the contained rows preprocessing technique, are also found by the clique and equal rows techniques combined.

Proof. Suppose that row t is contained in row s and that $K = \{k \in J \mid k \in R(s), k \notin R(t)\}$ is the set of columns that cover row s, but not row t. Following the contained rows preprocessing technique, all columns $k \in K$ and row s can be removed from the problem. However, this also means that all columns that cover row t have an element in common with every column $k \in K$. According to the clique rule, we can delete all columns $k \in K$. After this procedure, rows s and t are equal and we can delete one of them according to the equal rows preprocessing rule.

Although the clique rule dominates the contained rows rule, the latter can still be of value in a set partitioning algorithm. Table 2.6 shows the results of applying the contained rows, clique and equal rows techniques and the results of applying the clique and equal rows rules iteratively until no more reductions can be obtained. Applying the contained rows, clique and equal rows rules subsequently yields higher reductions than when we apply the clique and equal rows rules alone, while the computing time is lower. Moreover, the reductions of applying the contained rows, clique and equal rows rules subsequently are very close to the reductions of applying the clique and equal rows rules iteratively, while the computing times are much higher in the second case. The contained rows rule can thus be used to quickly remove the “easy” reduction, before applying the clique rule.

Table 2.6: The added value of the contained rows rule over the clique rule

			Contained Rows, Clique & Equal Rows					Clique & Equal Rows iteratively				
Name	**Cols**	**Rows**	**CR**	**RR**	**%CR**	**%RR**	**T**	**CR**	**RR**	**%CR**	**%RR**	**T**
nw41	197	17	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw32	294	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw40	404	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw08	434	24	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw15	467	31	58	0	12%	0%	0.00	58	0	12%	0%	0.00
nw21	577	25	5	0	1%	0%	0.00	5	0	1%	0%	0.00
nw22	619	23	12	0	2%	0%	0.00	12	0	2%	0%	0.00
nw12	626	27	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw39	677	25	2	0	0%	0%	0.00	2	0	0%	0%	0.00
nw20	685	22	30	0	4%	0%	0.00	30	0	4%	0%	0.00
nw23	711	19	93	0	13%	0%	0.00	93	0	13%	0%	0.00
nw37	770	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw26	771	23	123	2	16%	9%	0.00	126	2	16%	9%	0.00
nw10	853	24	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw34	899	20	39	0	4%	0%	0.00	39	0	4%	0%	0.00
Heart	926	180	42	44	5%	24%	0.02	42	44	5%	24%	0.03
nw43	1072	18	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw42	1079	23	118	0	11%	0%	0.00	118	0	11%	0%	0.00
Delta	1194	126	156	10	13%	8%	0.08	163	10	14%	8%	0.05
nw28	1210	18	384	0	32%	0%	0.00	384	0	32%	0%	0.02
nw25	1217	20	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw38	1220	23	232	0	19%	0%	0.00	232	0	19%	0%	0.00
nw27	1355	22	137	0	10%	0%	0.02	137	0	10%	0%	0.02
nw24	1366	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw35	1709	23	256	0	15%	0%	0.00	256	0	15%	0%	0.00
nw36	1783	20	265	0	15%	0%	0.00	265	0	15%	0%	0.00
Snowflake	2300	585	283	15	12%	3%	1.61	283	15	12%	3%	0.14
Fives	2440	72	0	0	0%	0%	0.02	0	0	0%	0%	0.02
Meteor	2464	60	437	0	18%	0%	0.03	472	0	19%	0%	0.05
nw29	2540	18	0	0	0%	0%	0.02	0	0	0%	0%	0.00
nw30	2653	26	10	0	0%	0%	0.02	10	0	0%	0%	0.00
nw31	2662	26	125	0	5%	0%	0.00	125	0	5%	0%	0.02
nw19	2879	40	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw33	3068	23	112	0	4%	0%	0.02	112	0	4%	0%	0.02
nw09	3103	40	0	0	0%	0%	0.00	0	0	0%	0%	0.02
nw07	5172	36	0	0	0%	0%	0.02	0	0	0%	0%	0.02
aa02	5198	531	1331	164	26%	31%	0.20	1350	169	26%	32%	0.52
nw06	6774	50	0	0	0%	0%	0.02	0	0	0%	0%	0.02
aa04	7195	426	1072	83	15%	19%	0.25	1072	83	15%	19%	0.67
aa06	7292	646	1345	137	18%	21%	0.39	1390	142	19%	22%	1.31
kl01	7479	55	905	8	12%	15%	0.05	905	8	12%	15%	0.08
aa05	8308	801	1933	251	23%	31%	0.55	2062	268	25%	33%	1.75
aa03	8627	825	1857	260	22%	32%	0.56	1920	272	22%	33%	1.78
nw11	8820	39	0	0	0%	0%	0.02	0	0	0%	0%	0.02
aa01	8904	823	1352	205	15%	25%	0.69	1360	206	15%	25%	1.72
nw18	10757	124	2	0	0%	0%	0.08	2	0	0%	0%	0.13
us02	13635	100	6686	55	49%	55%	0.24	6719	55	49%	55%	0.47
nw13	16043	51	0	0	0%	0%	0.05	0	0	0%	0%	0.05
us04	28016	163	17500	56	62%	34%	0.39	19784	62	71%	38%	1.27
kl02	36699	71	0	2	0%	3%	0.41	0	2	0%	3%	0.73
nw03	43749	59	0	0	0%	0%	0.16	0	0	0%	0%	0.17
nw01	51975	135	168	0	0%	0%	0.94	168	0	0%	0%	1.66
us03	85552	77	41850	26	49%	34%	2.66	41913	27	49%	35%	12.73
nw04	87482	36	0	0	0%	0%	0.31	0	0	0%	0%	0.31
nw02	87879	145	2	0	0%	0%	2.28	2	0	0%	0%	4.34
nw17	118607	61	0	0	0%	0%	0.61	0	0	0%	0%	0.58
nw14	123409	73	0	0	0%	0%	0.77	0	0	0%	0%	0.75
nw16	148633	139	0	0	0%	0%	2.92	0	0	0%	0%	2.66
nw05	288507	71	0	0	0%	0%	1.86	0	0	0%	0%	1.78
us01	1053137	145	86201	59	8%	41%	155.58	86201	59	8%	41%	318.49
Average	38585	123	2752	23	9%	6%	2.90	2797	24	9%	7%	5.91

Table 2.7: The added value of contained rows over the row combination heuristic

			RCT p = 0.0					Contained Rows, RCT p = 0.0				
Name	**Cols**	**Rows**	**CR**	**RR**	**%CR**	**%RR**	**T**	**CR**	**RR**	**%CR**	**%RR**	**T**
nw41	197	17	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw32	294	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw40	404	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw08	434	24	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw15	467	31	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw21	577	25	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw22	619	23	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw12	626	27	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw39	677	25	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw20	685	22	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw23	711	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw37	770	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw26	771	23	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw10	853	24	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw34	899	20	0	0	0%	0%	0.00	0	0	0%	0%	0.00
Heart	926	180	0	44	0%	24%	0.03	0	44	0%	24%	0.02
nw43	1072	18	0	0	0%	0%	0.00	0	0	0%	0%	0.02
nw42	1079	23	0	0	0%	0%	0.00	0	0	0%	0%	0.00
Delta	1194	126	0	10	0%	8%	0.02	0	10	0%	8%	0.02
nw28	1210	18	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw25	1217	20	0	0	0%	0%	0.02	0	0	0%	0%	0.00
nw38	1220	23	0	0	0%	0%	0.02	0	0	0%	0%	0.00
nw27	1355	22	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw24	1366	19	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw35	1709	23	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw36	1783	20	0	0	0%	0%	0.00	0	0	0%	0%	0.00
Snowflake	2300	585	265	15	12%	3%	0.45	265	15	12%	3%	1.52
Fives	2440	72	0	0	0%	0%	0.00	0	0	0%	0%	0.02
Meteor	2464	60	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw29	2540	18	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw30	2653	26	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw31	2662	26	0	0	0%	0%	0.02	0	0	0%	0%	0.00
nw19	2879	40	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw33	3068	23	0	0	0%	0%	0.00	0	0	0%	0%	0.02
nw09	3103	40	0	0	0%	0%	0.00	0	0	0%	0%	0.00
nw07	5172	36	0	0	0%	0%	0.02	0	0	0%	0%	0.02
aa02	5198	531	1242	165	24%	31%	0.50	1242	165	24%	31%	0.16
nw06	6774	50	0	0	0%	0%	0.00	0	0	0%	0%	0.02
aa04	7195	426	995	83	14%	19%	0.44	995	83	14%	19%	0.16
aa06	7292	646	1139	135	16%	21%	0.73	1139	135	16%	21%	0.23
kl01	7479	55	863	8	12%	15%	0.05	863	8	12%	15%	0.02
aa05	8308	801	1877	256	23%	32%	1.56	1877	256	23%	32%	0.45
aa03	8627	825	1577	259	18%	31%	1.78	1577	259	18%	31%	0.44
nw11	8820	39	0	0	0%	0%	0.00	0	0	0%	0%	0.02
aa01	8904	823	1217	203	14%	25%	1.55	1217	203	14%	25%	0.44
nw18	10757	124	0	0	0%	0%	0.02	0	0	0%	0%	0.03
us02	13635	100	2844	55	21%	55%	0.72	2844	55	21%	55%	0.13
nw13	16043	51	0	0	0%	0%	0.02	0	0	0%	0%	0.02
us04	28016	163	18576	62	66%	38%	1.44	18576	62	66%	38%	0.53
kl02	36699	71	0	2	0%	3%	0.17	0	2	0%	3%	0.11
nw03	43749	59	0	0	0%	0%	0.13	0	0	0%	0%	0.13
nw01	51975	135	0	0	0%	0%	0.16	0	0	0%	0%	0.14
us03	85552	77	36780	27	43%	35%	5.28	36780	27	43%	35%	1.44
nw04	87482	36	0	0	0%	0%	0.27	0	0	0%	0%	0.23
nw02	87879	145	0	0	0%	0%	0.30	0	0	0%	0%	0.28
nw17	118607	61	0	0	0%	0%	0.44	0	0	0%	0%	0.44
nw14	123409	73	0	0	0%	0%	0.41	0	0	0%	0%	0.42
nw16	148633	139	0	0	0%	0%	0.66	0	0	0%	0%	1.16
nw05	288507	71	0	0	0%	0%	1.13	0	0	0%	0%	1.22
us01	1053137	145	53729	59	5%	41%	156.59	53729	59	5%	41%	15.53
Average	38585	123	2018	23	4%	6%	2.91	2018	23	4%	6%	0.42

2.5.2 Relationship between the contained rows and row combinations techniques

Theorem 2.2 All reductions found by the contained rows preprocessing rule, will also be found by the row combination technique, for all non-negative values of p.

Proof. Suppose that row t is contained in row s and that $K = \{k \in J \mid k \in R(s), k \notin R(t)\}$ is the set of columns that cover row s, but not row t. Following the contained rows preprocessing technique, all columns $k \in K$ as well as row s can be removed from the problem. Now consider the row combination heuristic. We have:
$C(t,s) = \{j \in J(t) \mid j \notin J(s)\} = \emptyset$ and thus:
$f(t,s) = |C(t,s)| \cdot |C(s,t)| - |C(t,s)| - |C(s,t)| = -|C(s,t)| \leq 0$
Rows s and t will always be combined, since the value of f(t,s) is smaller than or equal to p% of the number of columns for all non-negative values of p. Therefore, all columns $k \in K$ and row s will be removed from the problem.

The added value of the contained rows preprocessing rule over the row combination technique (RCT) is illustrated by the results in Table 2.7. Performing the contained rows rule, followed by the row combination technique (p = 0.0), gives the same reductions for all instances as the row combination technique alone. However, the total computing time is much longer in the second case. Again, the contained rows preprocessing rule turns out to be a very fast procedure to take away the “easy” reductions before application of the more sophisticated row combination technique. Note that the reductions found after applying the contained rows and row combination techniques are greater than or equal to those found by the contained rows technique alone, while the computing times are comparable.

2.5.3 Relationship between the cut and clique rules

Theorem 2.3 All reductions achieved by the cut rule, will also be achieved by applying the clique rule.

Proof. Suppose that there is a set of three rows {r,s,t} and a row w, for which the following holds: row w is only covered by columns that cover at least two of the rows r, s and t. According to the cut preprocessing rule, we can now remove all columns that cover at least two of the rows r, s and t, but not row w. Consider such a column j. If we take this column in a solution, row w will be unsatisfiable. Therefore, according to the clique rule, column j can be removed from the problem.

Since the clique rule dominates the cut rule, we would expect the latter to be faster than the former in all instances. Actually, this is not the case and the cut rule is

even more time consuming on average than the clique rule, while the reductions found by the clique rule are much higher.

2.5.4 Relationship between k-rowset and clique

Theorem 2.4 For every value of k, the k-rowset preprocessing technique is a special case of the clique preprocessing technique.

Proof. Suppose that for $r_1,\ldots, r_k$, $k > 1$ there is no column j for which $r_1 \in R(j)$ and $r_2,\ldots, r_k \notin R(j)$. According to the k-rowset rule, all columns in $\{c \in J \mid r_1 \notin R(c), r_2,...,r_k \in R(c)\}$ can be removed from the problem. On the other hand, this also means that every column that covers row r_1 has at least one element in common with every column in $\{c \in J \mid r_1 \notin R(c), r_2,...,r_k \in R(c)\}$. Therefore, all columns in this set are also deleted by the clique rule.

Compared to the clique rule, the 3-rowset rule achieves fewer reductions in almost all cases. When applied iteratively, the clique rule achieves the most reductions for all instances, as expected, while the computing time of the clique rule is longer on average. In a preprocessing sequence, performing the contained rows, clique and equal rows rules subsequently has been proven to outperform the 3-rowset rule.

2.5.5 Relationship between the cut and 3-rowset rules

Theorem 2.5 All reductions achieved by the cut rule, will also be achieved by applying the 3-rowset rule.

Proof. Suppose that there is a set of three rows {r,s,t} and a row w, for which the following holds: row w is only covered by columns that cover at least two of the rows r, s and t. According to the cut preprocessing rule, we can now remove all columns that cover at least two of the rows r, s and t, but not row w. Consider such a column j and without loss of generality assume that j covers rows r and s. We can now delete column j according to the 3-rowset rule with rows w, r and s.

Although the 3-rowset rule dominates the cut rule, the computing time of the latter is much longer on our test set.

2.6 Combined computational results

As indicated before, the sequence in which preprocessing techniques are applied is

determinative for the overall success. Moreover, an important question is whether the preprocessing time needed outweighs the benefits in terms of decreased solution time. This section illustrates this with some computational experiments.

We compare the solution time of the well-known commercial solver CPLEX on the original problems with the time needed to solve the preprocessed problems. The calculations are made with the CPLEX 9.0 solver, used within the AIMMS modeling environment (Paragon, 2004). In order to make the comparison pure, we turned off the preprocessing option incorporated in CPLEX. We will discuss five different sequences of preprocessing techniques:

1. Equal columns, contained rows
2. Equal columns, contained rows, row combinations (p = 0.5)
3. Equal columns, contained rows, clique, equal rows
4. Equal columns, contained rows, clique, equal rows, row combinations (p = 0.5)
5. Equal columns, contained rows, row combinations (p = 0.5), clique, equal rows

All these sequences start with the equal columns rule, since this rule is very fast and powerful, as illustrated by the results discussed in Section 2.4.1. The contained rows rule is used next to remove the easy reductions, before the more time-consuming clique and row combination rules are applied. The results for these five sequences are summarized in Table 2.8.

For all sequences, a certain amount of column reduction is achieved for 59 out of the 60 instances. The amount of instances for which a positive row reduction is achieved, ranges from 15 to 40. The largest percentage column reduction is found for sequence 5, at 87%. The largest percentage row reduction is 71%, for sequences 2, 4 and 5. Total percentage column reduction, considered over all 60 instances, ranges from 45% to 47% over the sequences, while the total percentage row reduction ranges from 17% to 35%.

To measure the performance of the five preprocessing sequences, we compare the time of CPLEX on the original problems to the time of CPLEX on the preprocessed problems plus the preprocessing time. The difference is referred to as the ‘time benefit’. The total time over all 60 instances of CPLEX is equal to 2,147 seconds. For the best sequence, sequence 5, the time needed for preprocessing and solving the preprocessed problem is equal to 439 seconds. This means that the total time benefit equals to 1,708 seconds, or 80%. The lowest time benefit is still over 50%, for sequence 3.

Comparing sequences 1 and 2, we see that the row combination technique results in more row reductions, an increase in preprocessing time and a drastic decrease in total solution time. The same observations hold when we compare sequence 3 with sequence 4 and sequence 3 with sequence 5. The row combination technique is of great value to the preprocessing sequences. When comparing sequences 1 and 3, we see that adding the clique rule to the sequence does actually decrease the solution time of CPLEX by a small amount, although the increase in preprocessing time is higher. The value of the clique rule is illustrated by comparing

sequences 2 and 5. Adding the clique rule to sequence 2 leads to a substantial decrease in the total solution time. Note that the total solution time of CPLEX, with preprocessing turned on, on the 60 original problems, is equal to 1,209 seconds. With all five sequences, the total time of applying our preprocessing rules, plus the solution time of CPLEX on these preprocessed problems, is lower.

Table 2.8: Results for the five preprocessing sequences

	Original	Sequence 1	Sequence 2	Sequence 3	Sequence 4	Sequence 5
Number of instances	60	60	60	60	60	60
Number of instances with column reduction	0	59	59	59	59	59
Number of instances with row reduction	0	15	39	16	40	39
Largest % column reduction	0%	76%	86%	82%	86%	87%
Largest % row reduction	0%	55%	71%	55%	71%	71%
Total % column reduction	0%	45%	46%	47%	47%	47%
Total % row reduction	0%	17%	35%	19%	35%	35%
Total time preprocessing (s)	0	16.65	43.94	60.73	85.37	83.99
Total time CPLEX (s)	2146.59	990.34	545.61	973.46	499.36	355.14
Total time (s)	2146.59	1006.99	589.55	1034.19	584.73	439.13
Largest % time benefit CPLEX solver	0%	90%	94%	77%	89%	97%
Total % time benefit CPLEX solver	0.00%	53.09%	72.54%	51.82%	72.76%	79.54%

Within LaRSS, the calculation is started with preprocessing sequence 3 and the use of the row combination technique is postponed until knowledge about the lower- and upper bounds of the problem is available, since this reduces the amount of columns that are added to the tableau and greatly reduces the calculation time. When the costs of a new column are higher than the gap between the lower and upper bound, this column will never be in an optimal solution and does not have to be added to the tableau; see also Section 3.5.1. Just before branch and bound is started, the equal columns and clique rules are applied again to try to find more reductions. Chapter 7 will discuss the construction of LaRSS in more detail.

2.7 Concluding remarks

Preprocessing is a very powerful tool in reducing the solution time of set partitioning algorithms. The time needed to perform preprocessing is almost always compensated by the reduction in the solution time of the algorithm. By performing a sequence of preprocessing rules, the total solution time of our series of 60 test instances in CPLEX can be reduced by 80%. We have introduced two new preprocessing techniques for set partitioning problems: the row combination technique, which is a very powerful problem reduction tool, and the cut preprocessing rule, which turns out to be of less value. We have generalized two known preprocessing techniques, the equal columns and contained rows rules, and we have established many dominance relationships between the preprocessing techniques, which are illustrated by several computational experiments.

Chapter 3

Lower bounds

In any branching algorithm, the quality of the lower bound has a great influence on the computing time of the branching. Generally, a lower bound to a mathematical programming minimization problem is found by solving a relaxation of this problem. Since the relaxation is less constrained than the original problem, the value of the optimal solution of the original problem will never be below the value of the solution of the relaxation. Obviously, we want to find a relaxation that can be solved efficiently and that provides a good lower bound. Section 3.1 discusses the theoretical background of relaxation techniques and subgradient search. Section 3.2 discusses several subgradient search methods to solve the Lagrangian relaxation of the set partitioning problem. In Section 3.3 computational results considering these methods are reported and a compared. Section 3.4 deals with two dual heuristics to improve the lower bounds. Section 3.5 explores the role of the techniques implemented in LaRSS. Finally, we summarize our findings in Section 3.6.

3.1 Theoretical background

This section provides some theoretical background needed for the remainder of Chapter 3. We first discuss two alternative relaxation methods: linear programming relaxation and Lagrangian relaxation. Next, we introduce the concept of partial solutions and induced subproblems and discuss how to form lower bounds for these subproblems.

3.1.1 Linear programming relaxation

To obtain the linear programming (LP) relaxation of the set partitioning problem, we relax the integrality constraints. This results in the following problem:

$$z_{LP} = \min \sum_{j \in J} c_j \cdot x_j \qquad [3.1]$$

Subject to

$$\sum_{j \in J} a_{rj} \cdot x_j = 1 \qquad \forall r \in R \qquad [3.2]$$

$$x_j \geq 0 \qquad \forall j \in J \qquad [3.3]$$

The dual of this problem is given by:

$$z_{DLP} = \max \sum_{r \in R} u_r \qquad [3.4]$$

Subject to

$$\sum_{r \in R} a_{rj} \cdot u_r \leq c_j \qquad \forall j \in J \qquad [3.5]$$

u_r unrestricted $\forall r \in R$

We refer to the optimal solution of the linear programming relaxation as the LP lower bound (LBLP). If x^* is the optimal solution of the LP relaxation and u^* the optimal solution of the dual of the LP relaxation, then:

$$LBLP = \sum_{j \in J} c_j x_j^* = \sum_{r \in R} u_r^* \qquad [3.6]$$

The latter equality holds by the linear programming duality theorem.

3.1.2 Lagrangian relaxation

To obtain the Lagrangian relaxation of the set partitioning problem, the equality constraints are relaxed and taken into the objective with a Lagrangian multiplier λ_r:

$$z_{LR}(\lambda) = \min \sum_{j \in J} c_j \cdot x_j - \sum_{r \in R} \lambda_r \left(\sum_{j \in J} a_{rj} \cdot x_j - 1 \right) \qquad [3.7]$$

Subject to

$$x_j \in \{0,1\} \qquad \forall j \in J \qquad [3.8]$$

This can be rewritten to:

$$z_{LR}(\lambda) = \min \sum_{j \in J} \left(c_j - \sum_{r \in R} a_{rj} \lambda_r \right) \cdot x_j + \sum_{r \in R} \lambda_r \qquad [3.9]$$

Subject to [3.8].

Define the Lagrangian costs of a column j to be:

$$cl_j = c_j - \sum_{r \in R} a_{rj} \cdot \lambda_r \qquad [3.10]$$

Now the solution to the relaxed problem, given vector λ, is given by:

$$x_j = \begin{cases} 1 & \text{if } cl_j \leq 0 \\ 0 & \text{otherwise} \end{cases} \qquad [3.11]$$

The best lower bound we can find with this relaxation is given by:

$$LBLR = \max_{\lambda} z_{LR}(\lambda) \qquad [3.12]$$

Theorem 3.1 The value of the solution to the maximization problem given by [3.12] is equal to the value of the solution to the linear programming relaxation given by [3.1] – [3.3]:

$$\text{LBLR} = \max_{\lambda} \; z_{LR}(\lambda) = \sum_{j \in J} c_j x_j^* = \sum_{r \in R} u_r^* = \text{LBLP}$$

Proof. See Geoffrion (1974).

Since the maximization problem given by [3.12] is too time-consuming to solve to optimality, it is common practice to use heuristic methods to find a good value of the vector λ. Section 3.2 discusses different subgradient search methods.

3.1.3 Induced subproblems

A vector $x \in \{0,1\}^{|J|}$ such that:

$$\sum_{j \in J} a_{rj} \cdot x_j \leq 1 \qquad \forall r \in R \qquad [3.13]$$

is called a partial solution of the set partitioning problem with column set J. We define the set of rows covered by the partial solution to be R_1, and the set of rows that are not covered R_2. The union of R_1 and R_2 is equal to the total row set R and the two sets are disjoint. Furthermore, we define J_1 to be the set of all columns j for which x_j is equal to 1, $J_1 = \{j \in J \mid x_j = 1\}$ and J_3 the set of all columns k for which x_k is equal to zero that have at least one element in common with some $j \in J_1$, $J_3 = \left\{k \in J \setminus J_1 \mid \left(\cup_{j \in J_1} R(j)\right) \cap R(k) \neq \varnothing\right\}$. Now, $J_2 = J \setminus (J_1 \cup J_3)$ is the set of columns that can be chosen in the partial solution to cover the rows in R_2. The induced subproblem with row set R_2 and column set J_2 is again a set partitioning problem. During the branch and bound procedure, lower bounds for induced subproblems are of great interest.

3.1.4 Lower bounds for induced subproblems

The lower bound $LB(R_2,J_2)$ for the induced subproblem with row set R_2 and column set J_2 can be obtained by solving a new relaxation for the remaining problem on R_2 and J_2. However, during the branch and bound process, we consider thousands of partial solutions and solving a relaxation for each induced subproblem is rather time-consuming. Alternatively, any dual feasible vector u^f can be used to form lower bounds for the partial problem (Pierce and Lasky, 1973). If u^f is a feasible solution to the dual of the linear programming relaxation of the set partitioning problem then, for any induced subproblem, a lower bound $LB(R_2,J_2)$ is given by:

$$LB(R_2, J_2) = \sum_{r \in R_2} u_r^f \qquad [3.14]$$

A lower bound for the total problem, given the partial solution x, is now given by:

$$\sum_{j \in J} c_j \cdot x_j + \sum_{r \in R_2} u_r^f \qquad [3.15]$$

Another way to use this lower bound is with so-called reduced costs:

$$\sum_{j \in J} c_j \cdot x_j + \sum_{r \in R_2} u_r^f = \sum_{j \in J} \left(c_j - \sum_{r \in R} a_{rj} \cdot u_r^f \right) + \sum_{r \in R} u_r^f = \sum_{j \in J} cr_j \cdot x_j + \sum_{r \in R} u_r^f \qquad [3.16]$$

Where cr_j are the reduced costs of column j:

$$cr_j = c_j - \sum_r a_{rj} \cdot u_r^f \qquad [3.17]$$

This lower bounding mechanism can be powerful in the branching process, provided that tight lower- and upper bounds are available. Obviously, an optimal dual vector u^* constitutes the best possible dual feasible solution. However, the linear programming relaxation of a large set partitioning problem can be highly degenerate and high quality solvers are needed to solve them (Hoffman and Padberg, 1993). We therefore apply a Lagrangian relaxation to the set partitioning problem, followed by dual heuristics, to find a good dual feasible solution u^f.

3.1.5 Subgradient search

The most common way to solve the Lagrangian relaxation is an iterative method called subgradient search. This search techniques forms a sequence of vectors $\{\lambda^k\}_{k=0}^K$ that converges to a good solution of the problem [3.12]. Sections 3.2 and 3.3 discuss several subgradient search methods in detail. This section examines two general issues considering this technique: dual feasibility and convergence.

Generally, the vector λ that results from a subgradient search method like the methods discussed in Section 3.2, is not necessarily a feasible solution to the dual of the linear programming relaxation. As discussed in Section 3.1.4, dual feasibility of the solution is important to be able to calculate lower bounds for induced subproblems during the branch and bound procedure. Therefore, we apply a simple procedure to make the vector λ dual feasible. For a certain column j with negative Lagrangian costs cl_j, we reduce λ_r for the first row r that covers this column, with the amount $(-cl_j)$. In our experience, this adjustment hardly affects the bounds found.

Considering the convergence of subgradient search methods, we refer to the famous theorem of Polyak (1967):

Theorem 3.2 Let $\{\lambda^k\}_{k=0}^\infty$ be a sequence of Lagrangian multipliers for the problem given in [3.7] and [3.8], such that λ^0 is a start vector and for every k > 0:

$$\lambda^k = \lambda^{k-1} + s^k \cdot \frac{g^k}{\left\| g^k \right\|} \qquad [3.18]$$

with g^k the subgradient in the k^{th} iteration. If $\{s^k\}_{k=0}^{\infty}$ meets the following properties:

1. $s^k > 0 \ \forall \ k \in \{0,1,...\}$
2. $\lim_{k\to\infty} s^k = 0$
3. $\sum_{k=0}^{\infty} s^k = \infty$

then $\{\lambda^k\}_{k=0}^{\infty}$ will converge to $\arg\max_{\lambda} \ z_{LR}(\lambda)$.

Proof. See Polyak (1967).

In practice, methods that fulfill the requirements in theorem 3.2 and thus converge to the optimal solution, are extremely inefficient (Hunting, 1998). For none of the methods discussed in the next section, convergence to the optimal vector λ can be proved. However, all of these methods have been applied successfully in practice.

3.2 Subgradient search methods

This section discusses several subgradient search methods that are designed to find a good solution to the Lagrangian relaxation of the set partitioning problem.

3.2.1 Classic subgradient search

This section discusses the method of Held, Wolfe and Crowder (1974), applied to the Lagrangian relaxation of the set partitioning problem. We will refer to this method as the “classic subgradient search” method (CSS). The goal is to solve the problem [3.12] iteratively by determining a sequence of Lagrangian multipliers $\{\lambda^k\}_{k=0}^{K}$. To this end, we use the following iteration scheme:

$$\lambda_r^0 = \min_{j \in J(r)} \frac{c_j}{\sum_{t \in R} a_{tj}} \qquad [3.19]$$

$$\lambda_r^{k+1} = \lambda_r^k + \text{stepsize}^k \cdot g_r^k \qquad [3.20]$$

Here, the vector g^k represents the vector of subgradients and stepsize^k the stepsize used in the k^{th} iteration of the algorithm:

$$g_r^k = 1 - \sum_j a_{rj} \cdot x_j^k \qquad r \in R \qquad [3.21]$$

and

$$\text{stepsize}^k = \frac{C \cdot \left(\bar{z} - z_{LR}\left(\lambda^k\right)\right)}{\sqrt{\sum_r \left(g_r^k\right)^2}} \qquad [3.22]$$

The value of $\bar{z}$ is an overestimate of the optimal value [3.12]. In our implementation, we link the value of $\bar{z}$ to the value of our trivial lower bound, given by [3.19], in the following way:

$$\bar{z} = (1+y) \cdot \sum_{r \in R} \min_{j \in J(r)} \frac{c_j}{\sum_{t \in R} a_{tj}} \qquad [3.23]$$

We now have to determine the value of two parameters: y, $y \geq 0$ and C, $C \in (0,2]$.

The algorithm is stopped when the difference between two subsequent solutions is smaller than $\varepsilon = 0.01$.

3.2.2 Volume algorithm

Generally, the subgradient search method does not produce primal feasible solutions to the linear programming (LP) relaxation. Barahona and Anbil (2000, 2002) propose a method called the volume algorithm (VA) to solve the Lagrangian relaxation problem and to produce approximate solutions to the primal of the LP relaxation. We implemented this method as follows.

Step 0: Start with λ^0 as in [3.19] and solve problem [3.9] to get x^0 and $z^0 = z_{LR}(\lambda^0)$. Set t = 0 and $\bar{x} = x^0$.

Step 1: Define:

$$v_r^k = 1 - \sum_j a_{rj} \bar{x}_j \qquad [3.24]$$

$$\text{stepsize}^k = \beta \cdot \frac{\left(T - z^{k-1}\right)}{\sum_{r \in R} \left(v_r^{k-1}\right)^2} \qquad [3.25]$$

We now set $\bar{x} = \alpha \cdot x^k + (1-\alpha)\bar{x}$, where α, β and T are parameters, $\alpha \in (0,1]$, $\beta \in (0,2]$ and T is a target value, which we set very low at the start of the algorithm. Their values are determined as follows:

1. For α: Start with α^0. After every 100 iterations, we check whether $z_{LR}(\lambda)$ has increased by at least 1%. If this is not the case, we divide α by 2, unless α is smaller than 0.0001.
2. For β: Start with β^0. After 20 iterations without improvement we multiply by 0.66, as long as $\beta > 0.0005$. After iteration k we determine

   $$d^k = \sum_{r \in R} v_r^k \cdot \left(1 - \sum_{j \in J} a_{rj} \cdot x_j^k\right) \qquad [3.26]$$

   If $d^k \geq 0$, then we multiply β by 1.1. If $\beta > 2$, we set $\beta = 2$.

3. For T: Start with a value derived from the trivial lower bound given by [3.19]:

$$T = \frac{1}{2} \cdot \sum_{r \in R} \left(\min_{j \in J(r)} \frac{c_j}{\sum_{t \in R} a_{tj}} \right) \qquad [3.27]$$

If, in iteration k, $z_{LR}(\lambda_k) > 0.95 \cdot T$, we set $T = 1.05 \cdot z_{LR}(\lambda^k)$

Step 2: If we have not found a better lower bound in 100 iterations, we stop. Otherwise we set k = k+1 and go to step 1.

This algorithm has two parameters to set: α^0 and β^0. The resulting vector $\bar{x}$ gives an approximate primal solution to the LP relaxation.

3.2.3 Static convergent series

The method discussed here, referred to as the 'static convergence series' (SCS) method, is based upon the convergent series method of Goffin (1977), also discussed in Hunting (1998). The goal of the SCS search method is to determine a sequence of vectors $\{\lambda^k\}_{k=0}^{K}$ that converges to a good solution to the problem [3.12]. To this end, the following iteration scheme is used:

$$\lambda_r^0 = \min_{j \in J(r)} \frac{c_j}{\sum_{t \in R} a_{tj}} \qquad [3.28]$$

$$\lambda_r^{k+1} = \lambda_r^k + \text{stepsize}^k \cdot g_r^k \qquad [3.29]$$

Again, the vector g^k represents the vector of subgradients and stepsize^k the stepsize used in the k^{th} iteration of the algorithm:

$$g_r^k = 1 - \sum_j a_{rj} \cdot x_j^k \qquad r \in R \qquad [3.30]$$

and

$$\text{stepsize}^k = \frac{C^k}{\sqrt{\sum_r \left(g_r^k\right)^2}} \qquad [3.31]$$

C^k is determined by:

$$C^k = (\alpha)^k \cdot C^0 \qquad [3.32]$$

Since the speed of the subgradient search depends on the number of columns, we do not take all the columns into account at the start of the search. Instead, we only take the N_r columns with the lowest costs for every row. For this set of columns we perform the subgradient search. If the resulting λ gives the same lower bound for the whole set of columns as for the subset of columns, we keep this λ as the final solution. If this is not the case, we take a larger set of columns and start again. The maximum number of columns taken into account per row is given by:

$$N_r = \min\left(Q \cdot \frac{\sum_{s \in R} \sum_{j \in J} a_{sj}}{|J|} \;,\; |J(r)| \right) \qquad [3.33]$$

The search is stopped when the difference in solutions between two subsequent iterations is smaller than $\varepsilon = 0.01$. This method uses three parameters: α, C^0 and Q.

3.2.4 Dynamic convergent series

The dynamic convergent series (DCS) method is an extension of the static convergent series method discussed in 3.2.3. Considering the parameter α, we can say that the closer α is to 1, the better we expect the convergence to be, but the longer the method will take to converge. For this reason, we adjust the value of α during the subgradient search. In the beginning, when we are far from the optimal lower bound, we take a small value of α, $\alpha = \alpha_1$. While we come closer to the best lower bound, we adjust the value of α two times. This is done in the following way:

If $\left| \frac{z_{LR}(\lambda^{k+1}) - z_{LR}(\lambda^{k})}{z_{LR}(\lambda^{k+1})} \right| \leq 0.5$, then we set $\alpha = \alpha_2$ [3.34]

If $\left| \frac{z_{LR}(\lambda^{k+1}) - z_{LR}(\lambda^{k})}{z_{LR}(\lambda^{k+1})} \right| \leq 0.25$, then we set $\alpha = \alpha_3$ [3.35]

The moments at which we adjust the value of α are linked to the relative improvements of the lower bounds found; when the difference is smaller than 50% and 25% respectively, we adjust α. These numbers are determined by careful testing. The remaining parameters of this method are α_1, α_2, α_3, C^0 and Q.

3.2.5 Bundle dynamic convergent series

The bundle dynamic convergent series (BDCS) method is an extension of the dynamic convergent series method discussed in 3.2.4. In this revised version we use a so-called bundling technique to avoid the zigzag behavior that is typical for subgradient search methods (Byun, 2001). Instead of using a single subgradient at every iteration, we use a weighted combination of the current subgradient and the subgradients from previous iterations. This adjustment was originally proposed by Crowder (1976). We apply this technique to the DCS method discussed in Section 3.2.6. Instead of the subgradient vector g^k used in every iteration k, we use the adapted gradient vector $\tilde{g}^k$, where

$$\tilde{g}^k = h_1 \cdot g^k + h_2 \cdot g^{k-1} + h_3 \cdot g^{k-2} + h_4 \cdot g^{k-3} \qquad [3.36]$$

This adapted vector replaces g^k in [3.29]. The parameters of this method are h_1, h_2, h_3, h_4, α_1, α_2, α_3, C^0 and Q.

3.3 Computational results

This section reports the computational results for all methods discussed in Section 3.2 and compares their performance. All results are obtained by performing the method on the preprocessed problems in our test set, where we have applied equal columns, contained rows, clique and equal rows. This is the preprocessing sequence used in LaRSS before the lower bound calculation.

3.3.1 Classic subgradient search

Table 3.1 shows the lower bounds and computing times of the CSS method for four different combinations of the parameters y and C. On average, the computing time as well as the lower bounds found grow with the value of y. This also holds for parameter C; the larger C, the better the lower bound and the longer the computing time. Table 3.1 also shows the results of the linear programming relaxation, solved with CPLEX 9.0, with default settings. Compared to the LP results, the CSS performs poorly; the average lower bound is much worse, while the average computing time is longer for all values of the parameters.

3.3.2 Volume algorithm

Table 3.2 provides lower bounds and computing times of the VA for different values of α^0, with $\beta^0 = 1$. Table 3.3 shows the results for different values of β^0, with $\alpha^0 = 0.25$. On average, the lower bounds found, as well as the computing times of the volume algorithm, grow with the value of α^0. Comparing the results for β^0, the lower bounds are highest for $\beta^0 = 1$, while the computing times are also lowest for this value of β^0. Comparing the results of the VA to the LP results, we see that the bounds found are lower, while the average computing time is much longer. The time needed to perform the volume algorithm is too long for this algorithm to be of value in a solution algorithm.

3.3.3 Static convergent series

Table 3.4 shows the results of the SCS method for different values of α, with $C^0 = 1{,}000$ and $Q = 25$. As expected, the lower bound found by the subgradient search method increases with the value of α, as does the computing time of the method. As mentioned before, the lower bound found with any subgradient search method will never be higher than the linear programming bound. When $\alpha = 0.99875$, the Lagrangian relaxation bound is equal to the linear programming bound for 12 of the 60 instances. In 57 out of 60, the bound is less than 1% lower than the linear programming lower bound. For $\alpha = 0.975$, these numbers are 6 and 33, respectively.

Table 3.1 Results of the CSS method for different values of y and C

	CSS		CSS		CSS		CSS		LP	
	y = 0.5, C = 1.5		y = 0.5, C = 2		y = 1, C = 1.5		y = 1, C = 2			
Name	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**
nw41	10338.65	0.02	10471.73	0.03	10816.95	0.03	10415.07	0.05	10972.50	0.02
nw32	14076.25	0.02	14083.43	0.05	13917.17	0.06	13627.20	0.06	14570.00	0.03
nw40	7666.13	0.00	7821.10	0.05	10220.17	0.00	10225.29	0.03	10658.25	0.02
nw08	5999.09	0.00	5812.79	0.06	7632.76	0.00	7828.29	0.06	35894.00	0.03
nw15	58018.77	0.00	57818.76	0.09	63888.88	0.11	59588.64	0.11	67743.00	0.03
nw21	6041.35	0.00	6239.02	0.06	7160.41	0.09	7126.00	0.09	7380.00	0.01
nw22	4645.61	0.00	4618.01	0.06	6104.81	0.00	6103.65	0.05	6942.00	0.03
nw12	8090.62	0.00	8359.57	0.08	10512.86	0.05	11052.44	0.08	14118.00	0.03
nw39	8491.29	0.00	8781.61	0.05	9694.37	0.11	9642.05	0.13	9868.50	0.03
nw20	9991.47	0.27	10423.17	0.09	13269.19	0.08	13277.59	0.05	16626.00	0.03
nw23	7454.38	0.05	7796.40	0.03	9929.75	0.03	9931.18	0.03	12317.00	0.03
nw37	7102.96	0.00	7258.04	0.06	9477.16	0.05	9498.90	0.08	9961.50	0.03
nw26	4637.09	0.03	4783.67	0.09	6192.53	0.00	6206.09	0.08	6743.00	0.03
nw10	8944.28	0.00	10274.48	0.09	11881.92	0.08	12174.23	0.13	68271.00	0.03
nw34	6640.53	0.00	6512.24	0.13	8643.03	0.00	8882.90	0.06	10453.50	0.05
Heart	148.45	0.52	21.91	0.52	61.01	0.52	0.00	0.50	180.00	0.09
nw43	7845.05	0.08	8147.70	0.13	8680.73	0.17	7741.19	0.19	8897.00	0.05
nw42	6718.39	0.02	6761.34	0.13	6923.56	0.19	6380.94	0.20	7485.00	0.03
Delta	65.73	0.52	0.00	0.52	25.68	0.52	0.00	0.52	126.00	0.13
nw28	7073.29	0.08	7056.45	0.02	7437.00	0.00	7329.85	0.14	8169.00	0.03
nw25	3975.68	0.00	4019.13	0.13	5292.53	0.00	5293.86	0.06	5852.00	0.05
nw38	4647.75	0.11	4715.93	0.05	5479.09	0.20	5308.13	0.22	5552.00	0.05
nw27	5683.73	0.00	5849.25	0.16	7578.81	0.11	7629.17	0.14	9877.50	0.03
nw24	4427.74	0.17	4464.12	0.13	5741.89	0.17	5812.81	0.14	5843.00	0.03
nw35	5650.26	0.00	5648.28	0.14	7135.53	0.22	7054.01	0.25	7206.00	0.05
nw36	3615.84	0.02	3745.48	0.27	4814.62	0.00	4895.40	0.11	7260.00	0.05
Snowflake	0.00	0.02	2.29	0.00	0.00	0.02	2.29	0.02	14.85	0.89
Fives	5.89	0.77	0.00	0.78	2.07	0.80	0.00	0.77	12.00	0.09
Meteor	19.45	0.45	0.00	0.45	0.00	0.45	0.00	0.45	60.00	0.06
nw29	3300.38	0.27	3317.27	0.17	3897.07	0.42	3802.33	0.42	4185.33	0.05
nw30	3691.93	0.41	3630.79	0.42	3535.70	0.49	3419.82	0.52	3726.80	0.05
nw31	4369.73	0.30	4372.19	0.13	5818.42	0.00	5958.86	0.25	7980.00	0.05
nw19	3678.36	0.42	3607.74	0.47	4772.28	0.02	5115.79	0.58	10898.00	0.05
nw33	3735.11	0.00	3816.73	0.48	4948.77	0.02	4962.22	0.28	6484.00	0.06
nw09	10074.07	0.00	10757.94	0.50	13232.89	0.00	13529.09	0.34	67760.00	0.05
nw07	3662.17	0.02	3700.01	0.45	4872.09	0.02	4916.08	0.63	5476.00	0.06
aa02	19624.58	0.00	19921.97	0.91	25855.23	0.00	26032.69	0.80	30494.00	0.50
nw06	2798.88	0.02	2824.63	1.55	3740.54	0.02	3731.52	0.55	7640.00	0.16
aa04	17394.10	0.02	17512.67	1.45	23138.38	0.03	23168.95	1.03	25877.61	0.89
aa06	19169.82	0.02	19310.07	1.55	25422.57	0.02	25459.45	1.22	26977.19	1.06
kl01	831.75	0.02	803.21	1.44	1039.75	0.03	1040.17	0.66	1084.00	0.13
aa05	28845.79	1.58	29875.01	1.83	37582.40	0.03	37969.59	1.78	53735.93	1.72
aa03	27307.10	0.02	27521.94	1.92	36027.93	1.33	36737.48	1.83	49616.36	1.55
nw11	13819.43	0.02	14294.54	1.22	18524.20	0.03	18695.90	1.14	116254.50	0.11
aa01	29626.77	0.02	29729.44	2.19	39184.48	1.58	39383.86	2.09	55535.44	3.11
nw18	22754.19	1.66	23875.81	1.75	30592.95	0.03	31720.94	2.06	338864.25	0.49
us02	2466.92	0.02	2804.42	1.22	2863.09	0.02	2966.49	1.42	5965.00	0.13
nw13	11678.92	1.44	12283.63	2.02	15546.85	0.03	16009.87	1.92	50132.00	0.23
us04	7938.41	0.02	8443.60	1.20	10443.98	0.02	10800.53	1.31	17731.67	0.16
kl02	205.94	0.19	210.91	3.89	195.75	6.08	26.30	6.70	215.25	0.56
nw03	11366.36	0.17	11703.41	14.05	15152.05	10.05	15543.87	12.27	24447.00	0.94
nw01	84726.76	0.25	86248.10	13.75	112494.03	4.00	112494.34	3.50	114852.00	1.16
us03	2570.11	7.72	2790.36	7.22	3406.81	0.13	3406.16	9.38	5338.00	0.59
nw04	5350.19	0.31	5352.82	11.39	7132.61	1.30	7133.33	9.41	16310.67	1.25
nw02	79962.83	0.78	81120.78	87.66	105207.32	55.59	104792.24	70.27	105444.00	2.23
nw17	9599.74	35.86	9599.65	37.20	10178.06	85.27	9552.00	82.59	10875.75	2.05
nw14	13050.29	80.66	13280.40	87.66	17352.85	1.22	17701.20	91.63	61844.00	2.27
nw16	19056.37	1.38	22815.42	267.63	24027.84	167.67	27808.10	242.19	1181590.00	9.41
nw05	15930.42	148.75	15961.30	191.38	21123.79	4.00	21430.26	168.44	132878.00	4.59
us01	3226.58	3.23	3707.29	545.81	3919.27	3.47	4212.20	413.66	9963.07	14.19
Average	11663.83	4.81	11944.67	21.58	14829.07	5.78	14875.81	18.93	48653.81	0.87

Table 3.2 Results of the VA for different values of α^0 and $\beta^0 = 1$

	VA		VA		VA		LP	
	$\alpha_0 = 0.05$, $\beta_0 = 1$		$\alpha_0 = 0.1$, $\beta_0 = 1$		$\alpha_0 = 0.25$, $\beta_0 = 1$			
Name	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**
nw41	10966.39	0.02	10967.39	0.02	10970.33	0.02	10972.50	0.02
nw32	12555.15	0.02	14541.82	0.03	14545.92	0.02	14570.00	0.03
nw40	10538.38	0.02	10602.33	0.06	10651.77	0.03	10658.25	0.02
nw08	35584.31	0.02	35894.00	0.05	35894.00	0.03	35894.00	0.03
nw15	67056.40	0.03	67487.05	0.05	67723.13	0.05	67743.00	0.03
nw21	7336.90	0.05	7372.51	0.03	7376.51	0.05	7380.00	0.01
nw22	6774.60	0.03	6919.11	0.03	6939.20	0.03	6942.00	0.03
nw12	13349.76	0.02	14113.54	0.03	14117.91	0.05	14118.00	0.03
nw39	9867.47	0.05	9865.83	0.05	9867.39	0.05	9868.50	0.03
nw20	16386.64	0.05	16555.93	0.05	16608.08	0.05	16626.00	0.03
nw23	12063.63	0.03	12195.60	0.03	12313.53	0.05	12317.00	0.03
nw37	9925.25	0.05	9955.10	0.05	9959.45	0.03	9961.50	0.03
nw26	6654.22	0.05	6721.77	0.03	6740.08	0.05	6743.00	0.03
nw10	66461.34	0.05	62099.86	0.03	68225.28	0.06	68271.00	0.03
nw34	10378.29	0.05	10444.55	0.05	10450.69	0.06	10453.50	0.05
Heart	159.43	0.03	167.21	0.02	178.11	0.09	180.00	0.09
nw43	8858.41	0.06	8793.85	0.05	8890.66	0.08	8897.00	0.05
nw42	7411.74	0.09	7461.65	0.08	7479.91	0.06	7485.00	0.03
Delta	114.84	0.03	114.84	0.03	114.84	0.03	126.00	0.13
nw28	8129.32	0.05	8117.26	0.02	8167.45	0.05	8169.00	0.03
nw25	5794.88	0.06	5834.46	0.08	5849.34	0.08	5852.00	0.05
nw38	5529.52	0.09	5537.57	0.08	5548.01	0.06	5552.00	0.05
nw27	9839.21	0.05	9868.48	0.08	9520.83	0.02	9877.50	0.03
nw24	5804.46	0.06	5836.00	0.09	5841.31	0.09	5843.00	0.03
nw35	7194.73	0.11	7202.29	0.08	7204.18	0.09	7206.00	0.05
nw36	7129.78	0.09	7218.53	0.11	7252.71	0.13	7260.00	0.05
Snowflake	0.00	0.14	0.00	0.14	0.00	0.14	14.85	0.89
Fives	5.76	0.03	5.76	0.03	5.76	0.02	12.00	0.09
Meteor	59.40	0.03	59.40	0.03	59.40	0.02	60.00	0.06
nw29	3983.62	0.08	4065.37	0.06	4178.58	0.19	4185.33	0.05
nw30	3715.03	0.20	3717.97	0.20	3723.91	0.20	3726.80	0.05
nw31	7911.31	0.20	7969.33	0.16	7973.62	0.19	7980.00	0.05
nw19	10220.29	0.09	10898.00	0.22	10898.00	0.14	10898.00	0.05
nw33	6446.48	0.22	5987.41	0.05	6482.10	0.24	6484.00	0.06
nw09	66187.22	0.16	67352.16	0.20	67693.33	0.33	67760.00	0.05
nw07	5476.00	0.25	5476.00	0.20	5476.00	0.16	5476.00	0.06
aa02	28893.69	0.66	30309.13	0.69	30469.11	0.67	30494.00	0.50
nw06	7427.48	0.45	7602.15	1.08	7628.57	0.94	7640.00	0.16
aa04	24050.33	1.08	25116.76	1.34	25709.21	1.31	25877.61	0.89
aa06	25273.51	1.30	26489.69	1.47	26877.68	1.42	26977.19	1.06
kl01	1054.33	0.55	1075.39	0.74	1080.90	0.95	1084.00	0.13
aa05	51027.64	1.39	52821.71	1.80	53399.38	1.86	53735.93	1.72
aa03	47176.92	1.70	49006.32	1.95	49442.71	1.97	49616.36	1.55
nw11	101219.79	0.41	114102.05	0.61	115056.80	0.89	116254.50	0.11
aa01	51149.25	1.75	54200.80	2.20	55088.16	2.27	55535.44	3.11
nw18	318335.72	1.22	327366.35	1.25	335292.85	1.78	338864.25	0.49
us02	5897.80	1.59	5960.89	1.34	5963.46	0.91	5965.00	0.13
nw13	45970.26	0.89	49115.99	1.86	49667.28	1.74	50132.00	0.23
us04	17325.57	2.25	17606.08	2.02	17700.07	2.38	17731.67	0.16
kl02	210.77	2.20	213.63	2.25	214.65	2.42	215.25	0.56
nw03	23969.25	9.58	24326.69	10.55	24428.54	9.83	24447.00	0.94
nw01	111617.33	18.63	114011.50	18.08	114672.87	23.25	114852.00	1.16
us03	5261.63	18.99	5324.95	19.33	5338.00	18.19	5338.00	0.59
nw04	13738.99	12.36	14849.96	18.19	14259.77	16.91	16310.67	1.25
nw02	102198.77	53.61	104345.15	62.11	105361.44	63.31	105444.00	2.23
nw17	10604.01	55.27	10793.00	58.16	10857.08	66.06	10875.75	2.05
nw14	52513.30	33.20	54392.48	31.83	59033.51	66.39	61844.00	2.27
nw16	1060416.87	22.88	1067232.21	22.94	1147122.28	89.03	1181590.00	9.41
nw05	117829.94	76.77	128603.10	124.06	122722.75	107.78	132878.00	4.59
us01	9606.00	367.14	9848.87	517.28	9933.22	503.23	9963.07	14.19
Average	44977.32	11.47	46035.58	15.09	47704.03	16.47	48653.81	0.87

Table 3.3 Results of the VA for different values of β^0 and $\alpha^0 = 0.25$

	VA		VA		VA		LP	
	$\alpha_0 = 0.25, \beta_0 = 0.5$		$\alpha_0 = 0.25, \beta_0 = 1$		$\alpha_0 = 0.25, \beta_0 = 1.5$			
Name	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**
nw41	10972.44	0.02	10970.33	0.02	10970.87	0.02	10972.50	0.02
nw32	14565.79	0.02	14545.92	0.02	14561.50	0.03	14570.00	0.03
nw40	10649.96	0.03	10651.77	0.03	10654.58	0.03	10658.25	0.02
nw08	35231.66	0.02	35894.00	0.03	34541.13	0.02	35894.00	0.03
nw15	67683.30	0.05	67723.13	0.05	67706.19	0.05	67743.00	0.03
nw21	7376.54	0.05	7376.51	0.05	7377.34	0.05	7380.00	0.01
nw22	6938.99	0.05	6939.20	0.03	6938.14	0.05	6942.00	0.03
nw12	14117.93	0.05	14117.91	0.05	14117.48	0.05	14118.00	0.03
nw39	9866.65	0.05	9867.39	0.05	9862.78	0.05	9868.50	0.03
nw20	16604.88	0.05	16608.08	0.05	16611.02	0.05	16626.00	0.03
nw23	12314.13	0.03	12313.53	0.05	12164.92	0.02	12317.00	0.03
nw37	9960.09	0.03	9959.45	0.03	9958.64	0.05	9961.50	0.03
nw26	6742.23	0.05	6740.08	0.05	6740.51	0.05	6743.00	0.03
nw10	68102.25	0.08	68225.28	0.06	68007.02	0.06	68271.00	0.03
nw34	10450.81	0.05	10450.69	0.06	10451.17	0.05	10453.50	0.05
Heart	178.04	0.11	178.11	0.09	178.41	0.13	180.00	0.09
nw43	8892.95	0.08	8890.66	0.08	8892.11	0.06	8897.00	0.05
nw42	7480.50	0.08	7479.91	0.06	7480.88	0.09	7485.00	0.03
Delta	110.20	0.03	114.84	0.03	112.52	0.03	126.00	0.13
nw28	8167.02	0.06	8167.45	0.05	8113.61	0.03	8169.00	0.03
nw25	5850.83	0.06	5849.34	0.08	5849.51	0.08	5852.00	0.05
nw38	5550.39	0.08	5548.01	0.06	5550.19	0.08	5552.00	0.05
nw27	9875.21	0.06	9520.83	0.02	9865.48	0.06	9877.50	0.03
nw24	5840.43	0.08	5841.31	0.09	5841.78	0.08	5843.00	0.03
nw35	7202.54	0.06	7204.18	0.09	7181.45	0.05	7206.00	0.05
nw36	7255.47	0.13	7252.71	0.13	6234.43	0.03	7260.00	0.05
Snowflake	0.00	0.13	0.00	0.14	0.00	0.14	14.85	0.89
Fives	5.76	0.02	5.76	0.02	10.80	0.03	12.00	0.09
Meteor	57.00	0.02	59.40	0.02	58.20	0.03	60.00	0.06
nw29	4168.34	0.19	4178.58	0.19	4157.23	0.17	4185.33	0.05
nw30	3722.28	0.20	3723.91	0.20	3724.67	0.20	3726.80	0.05
nw31	7977.52	0.19	7973.62	0.19	7978.91	0.19	7980.00	0.05
nw19	10898.00	0.17	10898.00	0.14	10898.00	0.16	10898.00	0.05
nw33	6480.44	0.19	6482.10	0.24	6475.57	0.17	6484.00	0.06
nw09	67712.07	0.33	67693.33	0.33	67708.07	0.31	67760.00	0.05
nw07	5476.00	0.13	5476.00	0.16	5476.00	0.14	5476.00	0.06
aa02	30471.49	0.70	30469.11	0.67	30469.13	0.69	30494.00	0.50
nw06	7627.65	1.06	7628.57	0.94	7624.35	1.06	7640.00	0.16
aa04	25638.93	1.44	25709.21	1.31	25690.36	1.28	25877.61	0.89
aa06	26824.56	1.42	26877.68	1.42	26865.57	1.41	26977.19	1.06
kl01	1081.38	0.83	1080.90	0.95	1082.17	0.83	1084.00	0.13
aa05	53432.82	1.78	53399.38	1.86	53510.18	1.77	53735.93	1.72
aa03	49419.90	1.95	49442.71	1.97	49432.39	1.92	49616.36	1.55
nw11	107892.74	0.36	115056.80	0.89	115401.31	0.80	116254.50	0.11
aa01	54883.27	2.42	55088.16	2.27	55092.52	2.41	55535.44	3.11
nw18	331035.41	1.48	335292.85	1.78	333754.64	1.89	338864.25	0.49
us02	5965.00	1.34	5963.46	0.91	5965.00	1.27	5965.00	0.13
nw13	49517.78	2.02	49667.28	1.74	49807.58	1.77	50132.00	0.23
us04	17699.95	2.33	17700.07	2.38	17708.95	2.20	17731.67	0.16
kl02	214.56	2.23	214.65	2.42	214.57	2.44	215.25	0.56
nw03	24404.26	11.95	24428.54	9.83	24425.14	9.61	24447.00	0.94
nw01	114606.52	23.84	114672.87	23.25	114691.56	22.31	114852.00	1.16
us03	5338.00	18.58	5338.00	18.19	5338.00	18.94	5338.00	0.59
nw04	11103.85	18.38	14259.77	16.91	13383.50	15.41	16310.67	1.25
nw02	105362.33	69.06	105361.44	63.31	105370.21	61.28	105444.00	2.23
nw17	10858.50	58.81	10857.08	66.06	10849.85	64.02	10875.75	2.05
nw14	58760.49	66.66	59033.51	66.39	54108.88	40.56	61844.00	2.27
nw16	1048460.62	29.42	1147122.28	89.03	1064844.33	28.06	1181590.00	9.41
nw05	125333.21	106.55	122722.75	107.78	125036.25	125.63	132878.00	4.59
us01	9929.13	505.77	9933.22	503.23	9933.67	483.98	9963.07	14.19
Average	45839.02	15.56	47704.03	16.47	46217.52	14.91	48653.81	0.87

Table 3.4 Results of the SCS method for different values of α (C^0 = 1000, Q = 25)

	SCS		SCS		SCS		LP	
€	$\alpha = 0.95$, C^0 = 1000, Q = 25		$\alpha = 0.975$, C^0 = 1000, Q = 25		$\alpha = 0.99875$, C^0 = 1000, Q = 25			
Name	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**
nw41	10972.49	0.00	10972.47	0.00	10972.47	0.05	10972.50	0.02
nw32	14424.50	0.00	14565.43	0.00	14569.97	0.06	14570.00	0.03
nw40	10658.20	0.02	10658.20	0.02	10658.19	0.11	10658.25	0.02
nw08	10649.13	0.00	14572.61	0.00	35894.00	0.06	35894.00	0.03
nw15	55358.63	0.00	63235.61	0.02	67742.97	0.17	67743.00	0.03
nw21	7379.96	0.02	7379.96	0.02	7379.97	0.17	7380.00	0.01
nw22	6941.97	0.00	6941.96	0.02	6941.97	0.16	6942.00	0.03
nw12	14085.04	0.00	14118.00	0.00	14118.00	0.05	14118.00	0.03
nw39	9868.46	0.02	9868.48	0.00	9868.47	0.14	9868.50	0.03
nw20	15605.91	0.00	16525.36	0.00	16625.98	0.20	16626.00	0.03
nw23	11793.75	0.00	12317.00	0.00	12316.98	0.14	12317.00	0.03
nw37	9961.47	0.00	9961.45	0.00	9961.48	0.13	9961.50	0.03
nw26	6742.96	0.02	6742.98	0.00	6742.99	0.14	6743.00	0.03
nw10	17638.18	0.00	23464.19	0.00	68271.00	0.11	68271.00	0.03
nw34	10453.48	0.00	10453.46	0.00	10453.49	0.19	10453.50	0.05
Heart	126.23	0.02	176.06	0.03	179.41	0.66	180.00	0.09
nw43	8882.01	0.00	8896.94	0.00	8896.98	0.19	8897.00	0.05
nw42	7415.32	0.00	7484.96	0.02	7484.96	0.27	7485.00	0.03
Delta	1.19	0.02	7.22	0.03	125.47	0.72	126.00	0.13
nw28	7988.00	0.00	8168.95	0.00	8168.98	0.20	8169.00	0.03
nw25	5851.95	0.00	5851.96	0.02	5851.97	0.25	5852.00	0.05
nw38	5390.70	0.02	5526.47	0.02	5551.97	0.30	5552.00	0.05
nw27	9868.81	0.00	9877.47	0.02	9877.48	0.25	9877.50	0.03
nw24	5842.96	0.00	5842.96	0.02	5842.98	0.22	5843.00	0.03
nw35	7205.97	0.02	7205.98	0.02	7205.97	0.36	7206.00	0.05
nw36	6767.23	0.00	7112.89	0.02	7259.97	0.38	7260.00	0.05
Snowflake	0.00	0.14	0.00	0.27	0.00	5.66	14.85	0.89
Fives	11.60	0.03	11.71	0.06	11.67	1.11	12.00	0.09
Meteor	50.14	0.02	59.74	0.03	59.72	0.63	60.00	0.06
nw29	3953.07	0.00	4082.84	0.02	4185.30	0.27	4185.33	0.05
nw30	3714.07	0.02	3724.59	0.02	3726.75	0.44	3726.80	0.05
nw31	7571.65	0.00	7892.67	0.03	7979.97	0.42	7980.00	0.05
nw19	10493.14	0.02	10898.00	0.03	10898.00	0.33	10898.00	0.05
nw33	6471.78	0.00	6483.97	0.03	6483.96	0.45	6484.00	0.06
nw09	19856.74	0.02	26626.18	0.03	67760.00	0.58	67760.00	0.05
nw07	5476.00	0.02	5476.00	0.02	5476.00	0.22	5476.00	0.06
aa02	30329.62	0.05	30493.91	0.05	30493.92	0.97	30494.00	0.50
nw06	6840.67	0.03	7460.91	0.06	7639.89	2.39	7640.00	0.16
aa04	24543.66	0.08	25475.13	0.13	25871.67	2.42	25877.61	0.89
aa06	26848.29	0.06	26939.46	0.11	26975.75	2.03	26977.19	1.06
kl01	1082.93	0.06	1083.77	0.13	1083.88	1.94	1084.00	0.13
aa05	49802.41	0.08	51903.73	0.14	53723.16	2.00	53735.93	1.72
aa03	46954.48	0.08	48954.48	0.14	49614.82	4.28	49616.36	1.55
nw11	26649.91	0.03	33462.14	0.03	116254.49	0.92	116254.50	0.11
aa01	50254.14	0.09	52700.68	0.17	55507.69	2.88	55535.44	3.11
nw18	51531.79	0.09	67624.25	0.14	323137.25	4.72	338864.25	0.49
us02	5965.00	0.02	5965.00	0.05	5965.00	0.48	5965.00	0.13
nw13	23099.49	0.06	26778.92	0.08	50131.96	1.64	50132.00	0.23
us04	17488.32	0.05	17686.01	0.08	17731.55	1.50	17731.67	0.16
kl02	146.42	0.24	214.58	0.13	215.08	1.28	215.25	0.56
nw03	21531.30	0.31	23528.17	0.39	24446.96	2.66	24447.00	0.94
nw01	96438.15	0.25	99391.20	0.31	114852.00	4.09	114852.00	1.16
us03	5325.40	0.24	5338.00	0.31	5338.00	2.34	5338.00	0.59
nw04	7099.14	0.38	7728.13	0.70	16310.64	14.25	16310.67	1.25
nw02	91781.74	0.48	97838.95	0.61	105444.00	2.20	105444.00	2.23
nw17	10512.17	0.50	10750.18	0.59	10875.66	3.00	10875.75	2.05
nw14	23758.84	0.64	28891.11	0.78	61843.97	7.02	61844.00	2.27
nw16	115034.22	1.81	218718.26	2.53	1181590.00	27.55	1181590.00	9.41
nw05	29135.02	3.19	36861.62	3.38	132878.00	16.59	132878.00	4.59
us01	9794.25	3.95	9937.50	4.11	9962.26	18.56	9963.07	14.19
Average	18457.00	0.22	21648.51	0.27	48390.55	2.39	48653.81	0.87

Table 3.5 Results of the SCS method for different values of C^0 (α = 0.975, Q = 25)

	SCS		SCS		SCS		LP	
€	$\alpha = 0.975$, $C^0 = 500$, Q = 25		$\alpha = 0.975$, $C^0 = 1000$, Q = 25		$\alpha = 0.975$, $C^0 = 2500$, Q = 25			
Name	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**
nw41	10972.48	0.02	10972.47	0.00	10972.49	0.00	10972.50	0.02
nw32	14405.36	0.00	14565.43	0.00	14569.99	0.00	14570.00	0.03
nw40	10658.17	0.02	10658.20	0.02	10658.18	0.00	10658.25	0.02
nw08	10935.56	0.02	14572.61	0.00	24268.19	0.00	35894.00	0.03
nw15	55824.23	0.02	63235.61	0.02	67171.60	0.00	67743.00	0.03
nw21	7379.96	0.02	7379.96	0.02	7379.97	0.00	7380.00	0.01
nw22	6941.97	0.00	6941.96	0.02	6941.96	0.00	6942.00	0.03
nw12	14118.00	0.00	14118.00	0.00	14118.00	0.00	14118.00	0.03
nw39	9868.47	0.00	9868.48	0.00	9868.49	0.00	9868.50	0.03
nw20	15931.32	0.00	16525.36	0.00	16598.12	0.00	16626.00	0.03
nw23	11948.20	0.00	12317.00	0.00	12316.97	0.00	12317.00	0.03
nw37	9961.49	0.02	9961.45	0.00	9961.49	0.00	9961.50	0.03
nw26	6742.98	0.02	6742.98	0.00	6742.98	0.00	6743.00	0.03
nw10	18529.22	0.02	23464.19	0.00	37264.42	0.00	68271.00	0.03
nw34	10453.50	0.00	10453.46	0.00	10453.49	0.02	10453.50	0.05
Heart	175.88	0.02	176.06	0.03	163.36	0.03	180.00	0.09
nw43	8896.72	0.02	8896.94	0.00	8896.97	0.02	8897.00	0.05
nw42	7437.92	0.00	7484.96	0.02	7484.95	0.02	7485.00	0.03
Delta	70.19	0.03	7.22	0.03	0.00	0.05	126.00	0.13
nw28	7997.02	0.02	8168.95	0.00	8168.94	0.02	8169.00	0.03
nw25	5851.99	0.02	5851.96	0.02	5851.97	0.02	5852.00	0.05
nw38	5417.45	0.02	5526.47	0.02	5551.93	0.02	5552.00	0.05
nw27	9877.46	0.00	9877.47	0.02	9877.47	0.02	9877.50	0.03
nw24	5842.96	0.02	5842.96	0.02	5842.99	0.02	5843.00	0.03
nw35	7205.99	0.02	7205.98	0.02	7206.00	0.02	7206.00	0.05
nw36	6967.33	0.02	7112.89	0.02	7181.28	0.02	7260.00	0.05
Snowflake	0.00	0.27	0.00	0.27	0.00	0.25	14.85	0.89
Fives	11.67	0.05	11.71	0.06	11.61	0.06	12.00	0.09
Meteor	59.72	0.03	59.74	0.03	59.76	0.05	60.00	0.06
nw29	4000.03	0.02	4082.84	0.02	4138.55	0.02	4185.33	0.05
nw30	3724.42	0.02	3724.59	0.02	3725.68	0.03	3726.80	0.05
nw31	7592.41	0.02	7892.67	0.03	7979.51	0.03	7980.00	0.05
nw19	10779.70	0.03	10898.00	0.03	10898.00	0.03	10898.00	0.05
nw33	6483.92	0.03	6483.97	0.03	6483.95	0.03	6484.00	0.06
nw09	20417.54	0.03	26626.18	0.03	40711.47	0.03	67760.00	0.05
nw07	5476.00	0.02	5476.00	0.02	5476.00	0.02	5476.00	0.06
aa02	30398.47	0.06	30493.91	0.05	30493.93	0.08	30494.00	0.50
nw06	6895.89	0.06	7460.91	0.06	7638.25	0.13	7640.00	0.16
aa04	24680.09	0.11	25475.13	0.13	25782.65	0.13	25877.61	0.89
aa06	26896.65	0.11	26939.46	0.11	26893.60	0.11	26977.19	1.06
kl01	1083.75	0.11	1083.77	0.13	1083.78	0.33	1084.00	0.13
aa05	50214.28	0.13	51903.73	0.14	53310.65	0.14	53735.93	1.72
aa03	47456.65	0.13	48954.48	0.14	49434.52	0.14	49616.36	1.55
nw11	27498.07	0.05	33462.14	0.03	52040.91	0.05	116254.50	0.11
aa01	50667.33	0.16	52700.68	0.17	54328.61	0.19	55535.44	3.11
nw18	53963.56	0.14	67624.25	0.14	104258.57	0.28	338864.25	0.49
us02	5965.00	0.03	5965.00	0.05	5965.00	0.05	5965.00	0.13
nw13	23928.01	0.06	26778.92	0.08	33415.75	0.08	50132.00	0.23
us04	17535.87	0.06	17686.01	0.08	17725.76	0.08	17731.67	0.16
kl02	214.61	0.11	214.58	0.13	214.59	0.13	215.25	0.56
nw03	21830.95	0.41	23528.17	0.39	24424.15	0.41	24447.00	0.94
nw01	97276.86	0.31	99391.20	0.31	100534.54	0.33	114852.00	1.16
us03	5338.00	0.17	5338.00	0.31	5333.29	0.33	5338.00	0.59
nw04	7122.60	0.47	7728.13	0.70	8738.18	0.97	16310.67	1.25
nw02	92526.07	0.63	97838.95	0.61	101364.64	0.61	105444.00	2.23
nw17	10524.29	0.63	10750.18	0.59	10756.94	0.61	10875.75	2.05
nw14	24844.16	0.75	28891.11	0.78	35351.12	1.33	61844.00	2.27
nw16	121122.23	2.47	218718.26	2.53	486048.03	2.63	1181590.00	9.41
nw05	28946.82	3.38	36861.62	3.38	58249.98	3.45	132878.00	4.59
us01	9868.82	4.09	9937.50	4.11	9945.16	5.89	9963.07	14.19
Average	18762.60	0.26	21648.51	0.27	28472.16	0.32	48653.81	0.87

Table 3.6 Results of the SCS method for different values of Q (α = 0.975, C^0 = 1000)

	SCS		SCS		SCS		LP	
€	$\alpha = 0.975$, $C^0 = 1000$, Q = 25		$\alpha = 0.975$, $C^0 = 1000$, Q = 50		$\alpha = 0.975$, $C^0 = 1000$, Q = ∞			
Name	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**
nw41	10972.47	0.00	10972.47	0.00	10972.47	0.00	10972.50	0.02
nw32	14565.43	0.00	14565.43	0.00	14565.43	0.00	14570.00	0.03
nw40	10658.20	0.02	10658.20	0.02	10658.20	0.00	10658.25	0.02
nw08	14572.61	0.00	14572.61	0.02	14572.61	0.00	35894.00	0.03
nw15	63235.61	0.02	63235.61	0.02	63235.61	0.00	67743.00	0.03
nw21	7379.96	0.02	7379.96	0.02	7379.96	0.00	7380.00	0.01
nw22	6941.96	0.02	6941.96	0.02	6941.96	0.00	6942.00	0.03
nw12	14118.00	0.00	14118.00	0.00	14118.00	0.00	14118.00	0.03
nw39	9868.48	0.00	9868.48	0.00	9868.48	0.00	9868.50	0.03
nw20	16525.36	0.00	16525.36	0.02	16525.36	0.00	16626.00	0.03
nw23	12317.00	0.00	12317.00	0.00	12317.00	0.00	12317.00	0.03
nw37	9961.45	0.00	9961.45	0.00	9961.45	0.00	9961.50	0.03
nw26	6742.98	0.00	6742.98	0.02	6742.98	0.00	6743.00	0.03
nw10	23464.19	0.00	23464.19	0.02	23464.19	0.00	68271.00	0.03
nw34	10453.46	0.00	10453.46	0.02	10453.46	0.02	10453.50	0.05
Heart	176.06	0.03	176.06	0.03	175.04	0.03	180.00	0.09
nw43	8896.94	0.00	8896.97	0.02	8896.97	0.02	8897.00	0.05
nw42	7484.96	0.02	7484.96	0.02	7484.96	0.02	7485.00	0.03
Delta	7.22	0.03	7.22	0.05	18.37	0.03	126.00	0.13
nw28	8168.95	0.00	8168.95	0.00	8168.95	0.02	8169.00	0.03
nw25	5851.96	0.02	5851.96	0.02	5851.96	0.02	5852.00	0.05
nw38	5526.47	0.02	5526.47	0.02	5526.47	0.02	5552.00	0.05
nw27	9877.47	0.02	9877.47	0.00	9877.47	0.02	9877.50	0.03
nw24	5842.96	0.02	5842.96	0.02	5842.96	0.02	5843.00	0.03
nw35	7205.98	0.02	7205.98	0.03	7205.98	0.02	7206.00	0.05
nw36	7112.89	0.02	7112.89	0.02	7112.89	0.02	7260.00	0.05
Snowflake	0.00	0.27	0.00	0.25	0.00	0.25	14.85	0.89
Fives	11.71	0.06	11.62	0.06	11.62	0.05	12.00	0.09
Meteor	59.74	0.03	59.71	0.03	59.68	0.03	60.00	0.06
nw29	4082.84	0.02	4079.78	0.03	4079.78	0.03	4185.33	0.05
nw30	3724.59	0.02	3724.59	0.03	3724.59	0.05	3726.80	0.05
nw31	7892.67	0.03	7892.67	0.03	7892.67	0.03	7980.00	0.05
nw19	10898.00	0.03	10898.00	0.03	10898.00	0.03	10898.00	0.05
nw33	6483.97	0.03	6483.97	0.05	6483.97	0.05	6484.00	0.06
nw09	26626.18	0.03	26626.18	0.05	26626.18	0.03	67760.00	0.05
nw07	5476.00	0.02	5476.00	0.02	5476.00	0.03	5476.00	0.06
aa02	30493.91	0.05	30493.91	0.06	30493.91	0.06	30494.00	0.50
nw06	7460.91	0.06	7455.99	0.09	7459.67	0.14	7640.00	0.16
aa04	25475.13	0.13	25475.15	0.14	25475.15	0.13	25877.61	0.89
aa06	26939.46	0.11	26939.47	0.11	26939.47	0.11	26977.19	1.06
kl01	1083.77	0.13	1083.73	0.19	1083.75	0.11	1084.00	0.13
aa05	51903.73	0.14	51903.70	0.14	51903.68	0.14	53735.93	1.72
aa03	48954.48	0.14	48954.47	0.13	48954.47	0.14	49616.36	1.55
nw11	33462.14	0.03	33462.14	0.06	33462.14	0.11	116254.50	0.11
aa01	52700.68	0.17	52700.68	0.19	52700.68	0.19	55535.44	3.11
nw18	67624.25	0.14	67618.93	0.19	67618.93	0.20	338864.25	0.49
us02	5965.00	0.05	5965.00	0.06	5965.00	0.06	5965.00	0.13
nw13	26778.92	0.08	26896.34	0.09	26896.34	0.19	50132.00	0.23
us04	17686.01	0.08	17684.30	0.09	17684.30	0.09	17731.67	0.16
kl02	214.58	0.13	214.47	0.17	214.53	0.36	215.25	0.56
nw03	23528.17	0.39	23381.13	0.28	23372.11	0.83	24447.00	0.94
nw01	99391.20	0.31	99391.20	0.41	99391.20	0.66	114852.00	1.16
us03	5338.00	0.31	5338.00	0.23	5338.00	0.47	5338.00	0.59
nw04	7728.13	0.70	7727.30	0.55	7719.68	1.17	16310.67	1.25
nw02	97838.95	0.61	97838.95	0.73	97838.95	4.02	105444.00	2.23
nw17	10750.18	0.59	10747.53	0.69	10747.53	5.47	10875.75	2.05
nw14	28891.11	0.78	28774.74	0.92	28739.94	6.80	61844.00	2.27
nw16	218718.26	2.53	215449.10	3.42	210099.97	19.66	1181590.00	9.41
nw05	36861.62	3.38	37158.69	2.56	36619.75	13.75	132878.00	4.59
us01	9937.50	4.11	9940.44	4.36	9942.82	29.59	9963.07	14.19
Average	21648.51	0.27	21596.28	0.28	21497.56	1.42	48653.81	0.87

Table 3.5 shows the results of the subgradient search method for different values of C^0. These results are obtained with α = 0.975 and Q = 25. The lower bounds found by the subgradient search method increase with the value of C^0. For α = 0.975, the differences are quite large. However, when the value of α increases, the impact of the parameter C^0 diminishes. There is an obvious trade-off between these two parameters.

Table 3.6 shows the results of the subgradient search method for different values of parameter Q. These results are obtained with $\alpha = 0.975$ and C^0 = 1,000. When the value of Q is 25 or 50, this means that, especially for the large problems, we do not take the whole tableau into account in the subgradient search method. For comparison, we have also added the results for $C^0 = \infty$, meaning that we perform the subgradient search method for all columns. In this case, the computing times are longer than in the cases Q = 25 and Q = 50, while the values of the bounds are hardly affected. In some cases, the bound found with Q = 50 is even larger than in the case Q = ∞, which can be explained by small differences in the convergence of the method. The adjustment that we propose is valuable, since the computing time is decreased, while the bounds found are just as good.

3.3.4 Dynamic convergent series

The DCS method uses five parameters: $\alpha_1, \alpha_2, \alpha_3, C^0$ and Q. We illustrate the performance of this method by one combination of these parameters, based on testing and on the results of the SCS method discussed in Section 3.3.3. Table 3.7 shows results of the SCS method for $\alpha = 0.99875$, C^0 = 1,000 and Q = 25 as well as results of the DCS method with α_1 = 0.95, α_2 = 0.975, α_3 = 0.99875, C^0 = 1,000 and Q = 25 and the results of the LP relaxation. The bounds found with the DCS method are very close to those found by the SCS method, while the average computing time of the SCS is much longer. Compared to the linear programming method, the dynamic method is somewhat slower, while the bounds are lower by definition.

3.3.5 Bundle dynamic convergent series

Table 3.8 shows results for this method for different values of h_1, h_2, h_3, and h_4, applied on the preprocessed problems. The different combinations tested are:

1. h_1 = 0.6, h_2 = 0.2, h_3 = 0.1 and h_4 = 0.1
2. h_1 = 0.7, h_2 = 0.2, h_3 = 0.1 and h_4 = 0
3. h_1 = 0.8, h_2 = 0.2, h_3 = 0 and h_4 = 0

The first sequence was proposed by Byun (2001) for its robustness and good results. Comparing the results in Table 3.8, we see that the average computing time does not depend much on the combination of the weights. The more gradients we take into account, the longer the average computing time, although the differences are small.

Table 3.7 Results of the DCS method compared to SCS and LP

	SCS $\alpha = 0.99875$ $C^0 = 1000$, Q = 25		DCS $\alpha_1 = 0.95$, $a_2 = 0.975$, $a_3 = 0.99875$ $C^0 = 1000$, Q = 25		LP	
Name	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**
nw41	10972.47	0.05	10972.49	0.03	10972.50	0.02
nw32	14569.97	0.06	14569.98	0.08	14570.00	0.03
nw40	10658.19	0.11	10614.41	0.03	10658.25	0.02
nw08	35894.00	0.06	35894.00	0.03	35894.00	0.03
nw15	67742.97	0.17	67742.88	0.13	67743.00	0.03
nw21	7379.97	0.17	7379.82	0.13	7380.00	0.01
nw22	6941.97	0.16	6941.83	0.13	6942.00	0.03
nw12	14118.00	0.05	14118.00	0.03	14118.00	0.03
nw39	9868.47	0.14	9868.39	0.13	9868.50	0.03
nw20	16625.98	0.20	16611.41	0.09	16626.00	0.03
nw23	12316.98	0.14	12295.91	0.05	12317.00	0.03
nw37	9961.48	0.13	9961.48	0.13	9961.50	0.03
nw26	6742.99	0.14	6736.83	0.06	6743.00	0.03
nw10	68271.00	0.11	68271.00	0.08	68271.00	0.03
nw34	10453.49	0.19	10453.45	0.14	10453.50	0.05
Heart	179.41	0.66	178.77	0.56	180.00	0.09
nw43	8896.98	0.19	8896.87	0.14	8897.00	0.05
nw42	7484.96	0.27	7484.95	0.24	7485.00	0.03
Delta	125.47	0.72	124.58	0.63	126.00	0.13
nw28	8168.98	0.20	8168.97	0.19	8169.00	0.03
nw25	5851.97	0.25	5846.80	0.09	5852.00	0.05
nw38	5551.97	0.30	5538.01	0.13	5552.00	0.05
nw27	9877.48	0.25	9876.17	0.08	9877.50	0.03
nw24	5842.98	0.22	5842.96	0.22	5843.00	0.03
nw35	7205.97	0.36	7202.61	0.17	7206.00	0.05
nw36	7259.97	0.38	7258.75	0.23	7260.00	0.05
Snowflake	0.00	5.66	1.52	5.02	14.85	0.89
Fives	11.67	1.11	11.52	0.69	12.00	0.09
Meteor	59.72	0.63	60.00	0.48	60.00	0.06
nw29	4185.30	0.27	4183.05	0.16	4185.33	0.05
nw30	3726.75	0.44	3723.42	0.22	3726.80	0.05
nw31	7979.97	0.42	7979.97	0.41	7980.00	0.05
nw19	10898.00	0.33	10898.00	0.20	10898.00	0.05
nw33	6483.96	0.45	6483.95	0.39	6484.00	0.06
nw09	67760.00	0.58	67760.00	0.30	67760.00	0.05
nw07	5476.00	0.22	5476.00	0.14	5476.00	0.06
aa02	30493.92	0.97	30493.89	0.86	30494.00	0.50
nw06	7639.89	2.39	7637.64	1.16	7640.00	0.16
aa04	25871.67	2.42	25829.79	1.41	25877.61	0.89
aa06	26975.75	2.03	26956.63	1.36	26977.19	1.06
kl01	1083.88	1.94	1079.70	1.27	1084.00	0.13
aa05	53723.16	2.00	53717.31	1.53	53735.93	1.72
aa03	49614.82	4.28	49602.43	2.81	49616.36	1.55
nw11	116254.49	0.92	113768.78	0.38	116254.50	0.11
aa01	55507.69	2.88	55468.00	2.00	55535.44	3.11
nw18	323137.25	4.72	311565.89	2.39	338864.25	0.49
us02	5965.00	0.48	5965.00	0.33	5965.00	0.13
nw13	50131.96	1.64	50131.04	0.94	50132.00	0.23
us04	17731.55	1.50	17725.17	0.88	17731.67	0.16
kl02	215.08	1.28	214.24	1.08	215.25	0.56
nw03	24446.96	2.66	24446.95	2.56	24447.00	0.94
nw01	114852.00	4.09	114486.91	2.50	114852.00	1.16
us03	5338.00	2.34	5258.39	1.78	5338.00	0.59
nw04	16310.64	14.25	13875.40	8.88	16310.67	1.25
nw02	105444.00	2.20	105444.00	1.64	105444.00	2.23
nw17	10875.66	3.00	10874.92	2.13	10875.75	2.05
nw14	61843.97	7.02	61843.95	5.52	61844.00	2.27
nw16	1181590.00	27.55	1177343.20	21.58	1181590.00	9.41
nw05	132878.00	16.59	132874.54	11.27	132878.00	4.59
us01	9962.26	18.56	9956.06	13.45	9963.07	14.19
Average	48390.55	2.39	48033.14	1.69	48653.81	0.87

Table 3.8 Results of the BDCS method for different values of h_1, h_2, h_3 and h_4

Name	BDCS $h_1 = 0.6$ $h_2 = 0.2$ $h_3 = 0.1$ $h_4 = 0.1$ Bound	Time	BDCS $h_1 = 0.7$ $h_2 = 0.2$ $h_3 = 0.1$ $h_4 = 0$ Bound	Time	BDCS $h_1 = 0.8$ $h_2 = 0.2$ $h_3 = 0$ $h_4 = 0$ Bound	Time	LP Bound	Time
nw41	10967.48	0.03	10947.61	0.02	10972.20	0.05	10972.50	0.02
nw32	14141.93	0.09	14306.89	0.08	14374.87	0.06	14570.00	0.03
nw40	10657.02	0.08	10654.19	0.09	10654.99	0.08	10658.25	0.02
nw08	35893.09	0.22	35894.00	0.20	35894.00	0.16	35894.00	0.03
nw15	67742.99	0.19	67122.90	0.02	67742.97	0.19	67743.00	0.03
nw21	7096.34	0.14	7140.75	0.14	7371.13	0.11	7380.00	0.01
nw22	6874.24	0.17	6900.50	0.17	6908.26	0.16	6942.00	0.03
nw12	14091.11	0.11	14117.97	0.16	14073.34	0.05	14118.00	0.03
nw39	9867.41	0.19	9868.46	0.23	9868.46	0.20	9868.50	0.03
nw20	16612.64	0.19	16614.86	0.17	16607.46	0.11	16626.00	0.03
nw23	12312.67	0.11	12313.47	0.11	12316.24	0.11	12317.00	0.03
nw37	9961.44	0.13	9961.07	0.11	9961.31	0.11	9961.50	0.03
nw26	6733.41	0.09	6741.08	0.11	6519.97	0.03	6743.00	0.03
nw10	67588.94	0.41	67829.87	0.31	68259.80	0.27	68271.00	0.03
nw34	10453.48	0.20	10453.33	0.14	10453.12	0.13	10453.50	0.05
Heart	179.46	0.69	178.72	0.61	178.50	0.64	180.00	0.09
nw43	8896.15	0.16	8894.89	0.11	8881.58	0.09	8897.00	0.05
nw42	7482.28	0.17	7483.64	0.17	7481.67	0.14	7485.00	0.03
Delta	124.89	0.70	125.50	0.77	125.05	0.70	126.00	0.13
nw28	8167.77	0.19	8168.87	0.19	8167.83	0.09	8169.00	0.03
nw25	5802.89	0.20	5851.22	0.20	5850.82	0.17	5852.00	0.05
nw38	5551.57	0.22	5543.36	0.14	5539.40	0.13	5552.00	0.05
nw27	9876.15	0.23	9877.24	0.25	9876.32	0.19	9877.50	0.03
nw24	5201.80	0.22	5488.73	0.22	5832.88	0.11	5843.00	0.03
nw35	6189.19	0.34	7203.38	0.25	7205.06	0.27	7206.00	0.05
nw36	7246.60	0.36	7256.12	0.23	7254.24	0.22	7260.00	0.05
Snowflake	1.52	5.86	1.52	4.58	1.52	5.25	14.85	0.89
Fives	11.52	0.83	11.52	0.83	11.52	0.92	12.00	0.09
Meteor	60.00	0.34	60.00	0.50	60.00	0.53	60.00	0.06
nw29	4122.55	0.19	4160.84	0.17	4180.36	0.34	4185.33	0.05
nw30	3637.93	0.36	3726.24	0.33	3726.54	0.38	3726.80	0.05
nw31	7972.51	0.30	7969.61	0.25	7978.16	0.31	7980.00	0.05
nw19	10885.88	0.70	10887.36	0.70	10895.87	0.44	10898.00	0.05
nw33	6431.56	0.41	6482.83	0.33	6481.97	0.34	6484.00	0.06
nw09	66469.77	1.45	67505.53	1.61	67752.16	0.80	67760.00	0.05
nw07	5476.00	0.36	5476.00	0.56	5476.00	0.56	5476.00	0.06
aa02	30305.18	1.72	30383.46	1.97	30323.10	0.69	30494.00	0.50
nw06	7601.65	2.06	7599.76	1.55	7619.41	1.17	7640.00	0.16
aa04	25636.84	3.20	25758.91	3.00	25846.61	2.83	25877.61	0.89
aa06	26546.64	3.27	26717.13	3.02	26900.35	2.53	26977.19	1.06
kl01	1082.32	1.86	1083.26	1.77	993.26	1.16	1084.00	0.13
aa05	52184.68	3.56	53174.07	3.86	53512.56	3.64	53735.93	1.72
aa03	49000.46	3.59	49317.19	4.02	49489.17	7.30	49616.36	1.55
nw11	116103.58	1.14	116252.99	1.31	116024.99	1.05	116254.50	0.11
aa01	54791.60	4.45	55038.37	4.14	55248.01	4.47	55535.44	3.11
nw18	320464.78	5.64	318759.88	5.13	316103.89	5.11	338864.25	0.49
us02	5963.86	0.69	5965.00	0.80	5965.00	0.64	5965.00	0.13
nw13	48786.26	1.25	50020.70	1.77	50069.11	1.58	50132.00	0.23
us04	16328.92	3.03	17147.56	3.73	17307.74	3.23	17731.67	0.16
kl02	214.83	1.36	214.74	1.25	214.89	1.27	215.25	0.56
nw03	24322.47	4.11	24416.31	3.84	24422.62	3.19	24447.00	0.94
nw01	106737.51	10.30	110027.70	6.14	111403.60	4.36	114852.00	1.16
us03	5330.37	4.02	5337.75	3.80	5338.00	3.00	5338.00	0.59
nw04	14457.78	10.03	13677.06	10.17	15629.43	12.91	16310.67	1.25
nw02	101876.91	10.17	103710.58	12.64	105439.59	5.64	105444.00	2.23
nw17	9147.55	4.69	10145.70	4.92	10663.78	5.34	10875.75	2.05
nw14	59123.12	7.09	60942.74	6.34	61721.40	7.58	61844.00	2.27
nw16	1145115.25	29.44	1164809.56	31.64	1174952.61	33.88	1181590.00	9.41
nw05	131556.05	11.77	132094.51	12.14	132581.15	9.33	132878.00	4.59
us01	9914.86	15.44	9942.18	17.56	9957.76	17.64	9963.07	14.19
Average	47222.93	2.67	47762.62	2.69	48044.39	2.57	48653.81	0.87

Comparing the method to the original DCS results of Section 3.3.4, we see that the problems take more time on average to converge. The resulting lower bounds are higher in some cases, but not all, and the differences are small.

3.3.6 Comparison

This section compares the performance of all methods described in this chapter. Figure 3.1 gives an overview of the performance on two dimensions: quality of the lower bound and computing time. The quality of the lower bounds is measured by normalizing them against the optimal values. The results for the classic subgradient search method (CSS) are obtained from Table 3.1, for the volume algorithm (VA) from Tables 3.2 and 3.3, for the static convergent series method (SCS) from Tables 3.4 – 3.6, for the dynamic convergent series method (DCS) from Table 3.7 and finally, the results for the bundle dynamic convergent series method (BDCS) are obtained from Table 3.8. To give a general overview of methods, we have made no distinction between the different parameters.

Obviously, the best methods are located in the upper-left corner of the figure, with high bounds and low computing times. It is interesting to note that we can cluster the different methods. In the lower-right corner we see the method of Held et al. (1974), meaning that this method has long computing times and finds poor lower bounds on average. This method is developed several decades ago for the general subgradient search problem and is not sophisticated enough to compete with the other methods considered. The volume algorithm is concentrated in the upper-right corner of the diagram, meaning that this method can be used to obtain good lower bounds, but at long computing times. This is caused by the effort needed to maintain all the information during the calculations. In the lower-left corner, low calculating time and poor bounds, we see the static convergent series method. The best methods are located in the upper-left corner: the normal as well as the bundle dynamic convergent series method and the LP solver CPLEX. The SCS method is very fast, but gives poor quality bounds. If the value of α would be higher, the bounds would be better, but the computing times longer. The power of the DCS and BDCS is that they benefit from the speed of the algorithm with low values of α and from the quality of the algorithm with high values of α, by dynamically adjusting this value. The linear programming (LP) calculation of CPLEX performs very well, giving the best bounds at a relatively low computing time.

Note that in LaRSS the interaction between different techniques is very important, which leads to better lower bounds than presented in this section. This will be examined in Section 3.5.

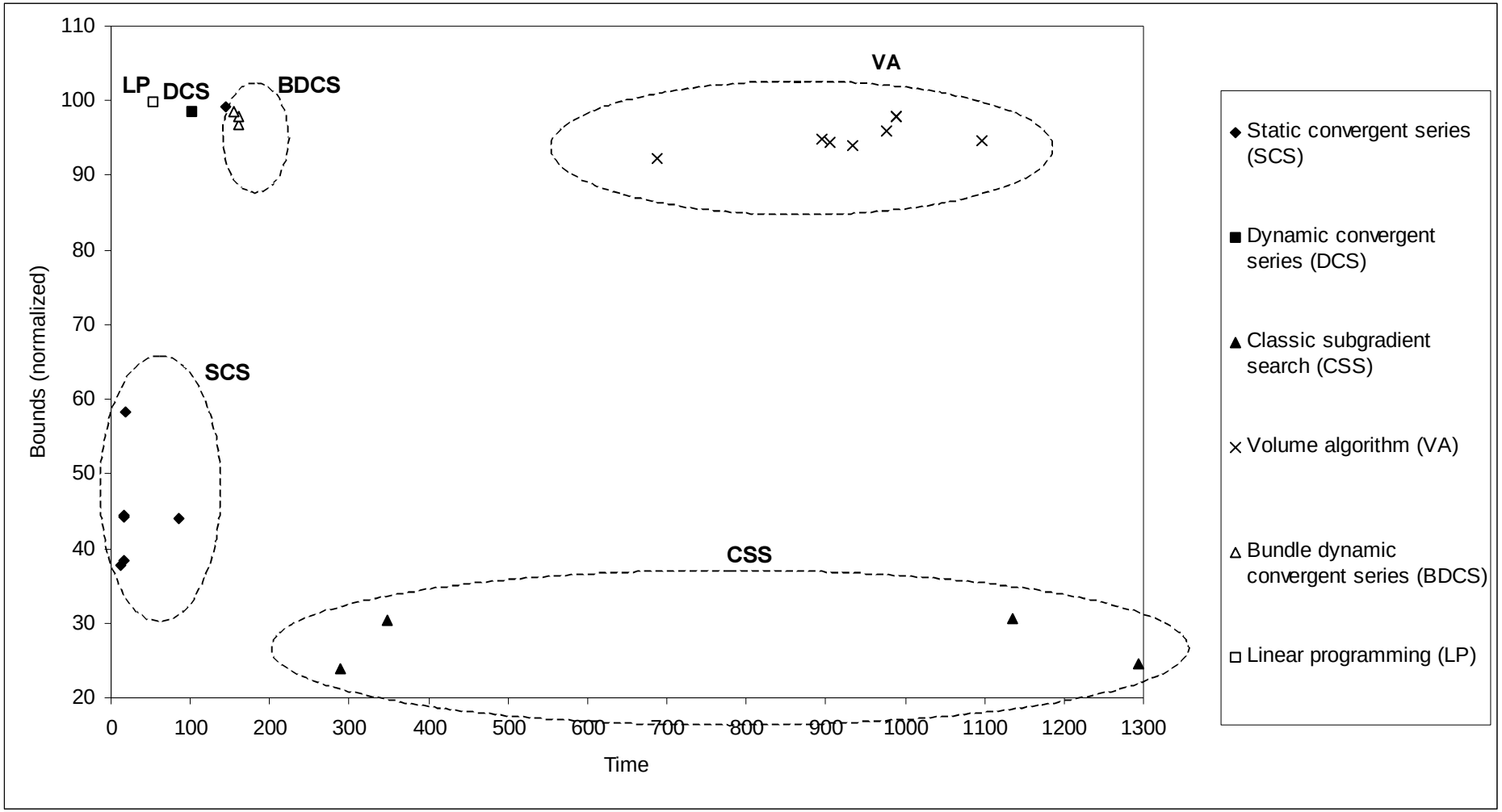


Figure 3.1: Comparison of different methods to determine lower bounds for the set partitioning problem

3.4 Dual Heuristics

This section discusses two dual heuristics to improve the lower bounds. Both heuristics start with a dual feasible vector $\{u^f\}_{r\in R}$

3.4.1 Simple improvement heuristic

If all columns j that cover row r have strictly positive reduced costs cr_j, then we can raise u_r^f with:

$$\Delta = \min_{j\in J(r)} cr_j \qquad [3.37]$$

After this step, the vector u^f still satisfies the dual feasibility constraint [3.5] and the lower bound is increased by Δ. We do this for all rows $r \in R$.

3.4.2 3OPT dual heuristic

The idea behind this heuristic stems from Fisher and Kedia (1986). The 3-opt heuristic is a local improvement heuristic that begins with a dual feasible solution and tries to improve this solution by simultaneously changing three u_r^f – values. Let u_{r1}^f, u_{r2}^f and u_{r3}^f be these three values. We now want to increase the value of the lower bound by simultaneously decreasing u_{r1}^f and increasing u_{r2}^f and u_{r3}^f, all by the same amount Δ. This will increase the value of the lower bound by Δ. This concept is implemented as follows.

Let J^b be the set of columns for which the constraints of the dual problem are binding:

$$J^b = \{j \in J \mid cr_j = 0\} \qquad [3.38]$$

We now have to make sure that two conditions are met to ensure feasibility of the resulting vector u^f:

$$\forall j \in J^b: \text{ if } j \in J(r_2) \text{ or } j \in J(r_3) \text{ then } j \in J(r_1) \qquad [3.39]$$

$$\nexists j \in J^b : j \in J(r_2) \text{ and } j \in J(r_3) \qquad [3.40]$$

Note that the proposed improvement is allowed if and only if these two conditions are met. If we have found three rows r_1, r_2 and r_3 for which the conditions hold, we determine the maximum allowed value of Δ such that the constraints in [3.5] hold for all columns and the resulting vector remains dual feasible. The heuristic consists of a complete search of all combinations of three rows in the problem. Although further improvements are possible when this method is applied iteratively, we apply it only once for every possible combination of three rows, since the extra improvements are too small relative to the extra computing times.

3.4.3 Computational results

Table 3.9: Computational results of the dual heuristics

	DCS		Update heuristic		3OPT heuristic	
Name	Bound	Time	Bound	Time	Bound	Time
nw41	10972.49	0.03	10972.49	0.00	10972.50	0.00
nw32	14569.98	0.08	14569.99	0.00	14570.00	0.00
nw40	10614.41	0.03	10641.59	0.00	10647.68	0.00
nw08	35894.00	0.03	35894.00	0.00	35894.00	0.00
nw15	67742.88	0.13	67743.00	0.00	67743.00	0.00
nw21	7379.82	0.13	7379.99	0.00	7380.00	0.00
nw22	6941.83	0.13	6941.94	0.00	6942.00	0.00
nw12	14118.00	0.03	14118.00	0.00	14118.00	0.00
nw39	9868.39	0.13	9868.39	0.00	9868.50	0.00
nw20	16611.41	0.09	16621.65	0.00	16621.65	0.00
nw23	12295.91	0.05	12315.39	0.00	12316.37	0.00
nw37	9961.48	0.13	9961.48	0.00	9961.50	0.00
nw26	6736.83	0.06	6739.55	0.00	6743.00	0.00
nw10	68271.00	0.08	68271.00	0.00	68271.00	0.00
nw34	10453.45	0.14	10453.49	0.00	10453.50	0.00
Heart	178.77	0.56	179.51	0.00	179.74	0.00
nw43	8896.87	0.14	8896.89	0.00	8896.93	0.00
nw42	7484.95	0.24	7484.98	0.00	7485.00	0.00
Delta	124.58	0.63	125.59	0.00	125.67	0.02
nw28	8168.97	0.19	8168.97	0.00	8169.00	0.00
nw25	5846.80	0.09	5850.22	0.00	5852.00	0.00
nw38	5538.01	0.13	5548.29	0.00	5549.22	0.00
nw27	9876.17	0.08	9877.31	0.00	9877.50	0.00
nw24	5842.96	0.22	5842.99	0.00	5843.00	0.00
nw35	7202.61	0.17	7204.45	0.00	7206.00	0.00
nw36	7258.75	0.23	7258.93	0.00	7259.73	0.00
Snowflake	1.52	5.02	11.22	0.00	11.52	12.02
Fives	11.52	0.69	11.72	0.02	11.72	0.00
Meteor	60.00	0.48	60.00	0.00	60.00	0.00
nw29	4183.05	0.16	4183.93	0.00	4183.96	0.00
nw30	3723.42	0.22	3723.80	0.00	3724.02	0.00
nw31	7979.97	0.41	7979.98	0.00	7980.00	0.00
nw19	10898.00	0.20	10898.00	0.00	10898.00	0.00
nw33	6483.95	0.39	6483.97	0.02	6484.00	0.00
nw09	67760.00	0.30	67760.00	0.00	67760.00	0.00
nw07	5476.00	0.14	5476.00	0.00	5476.00	0.00
aa02	30493.89	0.86	30494.00	0.00	30494.00	0.08
nw06	7637.64	1.16	7638.22	0.00	7639.24	0.00
aa04	25829.79	1.41	25852.93	0.00	25860.11	0.06
aa06	26956.63	1.36	26965.83	0.00	26966.51	0.14
kl01	1079.70	1.27	1082.98	0.00	1083.49	0.00
aa05	53717.31	1.53	53720.45	0.00	53720.81	0.19
aa03	49602.43	2.81	49611.04	0.00	49611.61	0.20
nw11	113768.78	0.38	113823.54	0.00	113828.63	0.00
aa01	55468.00	2.00	55476.31	0.00	55478.13	0.23
nw18	311565.89	2.39	311574.08	0.00	311578.36	0.02
us02	5965.00	0.33	5965.00	0.00	5965.00	0.00
nw13	50131.04	0.94	50131.76	0.00	50131.96	0.00
us04	17725.17	0.88	17729.71	0.00	17729.89	0.02
kl02	214.24	1.08	214.86	0.00	214.97	0.00
nw03	24446.95	2.56	24446.98	0.02	24447.00	0.02
nw01	114486.91	2.50	114491.63	0.03	114492.97	0.03
us03	5258.39	1.78	5313.36	0.02	5331.94	0.03
nw04	13875.40	8.88	13878.78	0.03	13878.86	0.03
nw02	105444.00	1.64	105444.00	0.08	105444.00	0.08
nw17	10874.92	2.13	10875.08	0.08	10875.16	0.08
nw14	61843.95	5.52	61844.00	0.08	61844.00	0.09
nw16	1177343.20	21.58	1177345.81	0.14	1177697.09	0.16
nw05	132874.54	11.27	132878.00	0.24	132878.00	0.17
Us01	9956.06	13.45	9960.94	0.39	9961.24	0.55
Average	48033.14	1.69	48037.97	0.02	48044.81	0.24

Table 3.9 shows the computational results of the dual heuristics on our test set. As a starting point, we consider the lower bounds resulting from the dynamic convergent series method with $\alpha_1 = 0.95$, $\alpha_2 = 0.975$, $\alpha_3 = 0.99875$, $C^0 = 1{,}000$ and $Q = 25$. To these bounds, we first apply the simple update heuristic and then the 3OPT heuristic. Table 3.9 shows the bounds after each of these steps and the computing times of the heuristics. Both heuristics are performed rapidly and the increase in lower bounds is quite large in some instances.

3.5 Lower bounds in LaRSS

In LaRSS, the interaction between the methods used is very important for the performance of the individual methods. Use of a combination of techniques renders the lower bounds as well as the computing time comparable to the performance of CPLEX. This section introduces reduced cost fixing, explains the use of the lower bounding procedures discussed in Section 3.2 in LaRSS and presents the results of these techniques within LaRSS. For a complete discussion of the composition of LaRSS, see to Chapter 7.

3.5.1 Reduced cost fixing

Given a set partitioning problem with constraint set R and a set of columns J, suppose that an upper bound UB and a dual feasible solution to the dual of the LP relaxation $\{u_r^f\}_{r\in R}$ exists, such that a lower bound LB to the problem is given by:

$$LB = \sum_{r\in R} u_r^f \qquad [3.41]$$

For every column $j \in J$ we know that, when a partial solution is formed by {j}, a lower bound to the induced subproblem is given by:

$$\sum_{r\in R} u_r^f + cr_j \qquad [3.42]$$

If the lower bound on the induced subproblem exceeds the upper bound UB, we know that the partial solution, consisting of column j, can never be extended to an optimal solution. Therefore, we can say that if:

$$cr_j > UB - \sum_{r\in R} u_r^f \qquad [3.43]$$

then column j will not be part of an optimal solution and x_j can be fixed to 0. This procedure is called reduced cost fixing.

3.5.2 Methods used in LaRSS for determination of lower bounds

In LaRSS, the following sequence of methods is applied to determine the lower bounds used in the branch and bound routine:

1. Preprocessing: equal columns, contained rows, clique and equal rows (Chapter 2)
2. Lagrangian relaxation: static convergent series method (Section 3.2.3) with $\alpha = 0.95$, $C^0 = 1{,}000$ and $Q = 25$
3. Dual heuristics: simple improvement and 3OPT heuristics (Section 3.4)
4. Primal heuristic to determine upper bound (Chapter 4)
5. Reduced cost fixing (Section 3.5.1)
6. Preprocessing: row combination technique with $p = 0.5$
7. Lagrangian relaxation: dynamic convergent series method (Section 3.2.4) with $\alpha_1 = 0.95$, $\alpha_2 = 0.975$, $\alpha_3 = 0.99875$, $C^0 = 1{,}000$ and Q resulting from 2.
8. Dual heuristics: simple improvement and 3OPT heuristics (Section 3.4)

The Lagrangian relaxation is solved twice: once using the SCS method, and once using the DCS method. The first time we aim at finding a lower bound rapidly, where a short computation time is more important than a high quality bound. Together with the primal heuristic and the reduced cost fixing procedure, this method is used to clear away part of the columns before the row combination heuristic starts. The row combination heuristic is used to improve the formulation of the set partitioning problem before the second Lagrangian relaxation is solved by the dynamic convergent series method. This time the aim is indeed to find a high quality lower bound. Using the row combination technique before the DCS allows the lower bounds found to be higher than the optimal solution of the LP relaxation.

3.5.3 Lower bound results of LaRSS

Table 3.10 shows the lower bounds that are calculated by LaRSS with the procedures described in Section 3.5.2. The time reported in the table is the total time of all lower bound techniques, i.e. the SCS and DCS methods and the dual heuristics. Table 3.10 also shows the LP results of CPLEX 9.0 on the preprocessed problems, as well as on the original test set. The preprocessing techniques used here do not influence the value of the solution of the LP relaxation, but they do influence the average computing time. On average, the LP bounds are better than the bounds found by LaRSS, although in 36 out of 60 times the lower bounds of LaRSS are higher than the LP bounds. Comparing the computing time of LaRSS to the time of CPLEX on the preprocessed problems, CPLEX is faster on average and in 33 out of 60 instances. Comparing the performance of LaRSS to CPLEX on the original problems, LaRSS is faster on average and in 28 out of 60 problems. The performance of LaRSS is comparable to the performance of CPLEX on the account of computing time as well as quality of the lower bounds.

Table 3.10: Results of LaRSS compared to the LP results

	LaRSS		LP after preprocessing		LP	
Name	**Bound**	**Time**	**Bound**	**Time**	**Bound**	**Time**
nw41	11307.00	0.00	10972.50	0.02	10972.50	0.02
nw32	14570.00	0.03	14570.00	0.03	14570.00	0.01
nw40	10658.09	0.02	10658.25	0.02	10658.25	0.03
nw08	35894.00	0.02	35894.00	0.03	35894.00	0.01
nw15	67743.00	0.00	67743.00	0.03	67743.00	0.03
nw21	7408.00	0.00	7380.00	0.01	7380.00	0.03
nw22	6984.00	0.00	6942.00	0.03	6942.00	0.03
nw12	14118.00	0.00	14118.00	0.03	14118.00	0.03
nw39	9868.50	0.02	9868.50	0.03	9868.50	0.03
nw20	16624.72	0.03	16626.00	0.03	16626.00	0.02
nw23	12317.00	0.03	12317.00	0.03	12317.00	0.02
nw37	10068.00	0.02	9961.50	0.03	9961.50	0.03
nw26	6796.00	0.00	6743.00	0.03	6743.00	0.03
nw10	68271.00	0.09	68271.00	0.03	68271.00	0.03
nw34	10453.50	0.02	10453.50	0.05	10453.50	0.03
Heart	179.54	0.64	180.00	0.09	180.00	0.08
nw43	8904.00	0.00	8897.00	0.05	8897.00	0.05
nw42	7484.98	0.06	7485.00	0.03	7485.00	0.03
Delta	126.00	0.28	126.00	0.13	126.00	0.13
nw28	8298.00	0.02	8169.00	0.03	8169.00	0.03
nw25	5852.00	0.06	5852.00	0.05	5852.00	0.03
nw38	5550.87	0.02	5552.00	0.05	5552.00	0.05
nw27	9933.00	0.02	9877.50	0.03	9877.50	0.05
nw24	6314.00	0.02	5843.00	0.03	5843.00	0.03
nw35	7206.00	0.03	7206.00	0.05	7206.00	0.05
nw36	7259.96	0.09	7260.00	0.05	7260.00	0.05
Snowflake	11.96	7.98	14.85	0.89	14.85	0.92
Fives	11.99	1.05	12.00	0.09	12.00	0.08
Meteor	60.00	0.25	60.00	0.06	60.00	0.06
nw29	4189.80	0.08	4185.33	0.05	4185.33	0.06
nw30	3723.43	0.06	3726.80	0.05	3726.80	0.05
nw31	7980.00	0.08	7980.00	0.05	7980.00	0.06
nw19	10898.00	0.03	10898.00	0.05	10898.00	0.06
nw33	6678.00	0.02	6484.00	0.06	6484.00	0.06
nw09	67760.00	0.28	67760.00	0.05	67760.00	0.06
nw07	5476.00	0.02	5476.00	0.06	5476.00	0.08
aa02	30494.00	0.78	30494.00	0.50	30494.00	0.64
nw06	7639.78	0.45	7640.00	0.16	7640.00	0.17
aa04	25870.36	1.95	25877.61	0.89	25877.61	1.27
aa06	26973.26	1.78	26977.19	1.06	26977.19	1.28
kl01	1083.61	0.23	1084.00	0.13	1084.00	0.20
aa05	53721.42	1.77	53735.93	1.72	53735.93	2.25
aa03	49607.10	1.49	49616.36	1.55	49616.36	2.24
nw11	112403.86	0.30	116254.50	0.11	116254.50	0.14
aa01	55519.00	2.45	55535.44	3.11	55535.44	4.56
nw18	328735.44	2.42	338864.25	0.49	338864.25	0.41
us02	5965.00	0.06	5965.00	0.13	5965.00	0.38
nw13	50131.31	0.70	50132.00	0.23	50132.00	0.27
us04	17729.56	0.30	17731.67	0.16	17731.67	0.73
kl02	215.05	0.80	215.25	0.56	215.25	0.69
nw03	24447.00	1.91	24447.00	0.94	24447.00	0.88
nw01	114852.00	2.87	114852.00	1.16	114852.00	0.94
us03	5338.00	0.19	5338.00	0.59	5338.00	2.14
nw04	16310.18	4.56	16310.67	1.25	16310.67	1.58
nw02	105444.00	2.48	105444.00	2.23	105444.00	1.75
nw17	10874.03	1.83	10875.75	2.05	10875.75	2.31
nw14	61844.00	5.05	61844.00	2.27	61844.00	2.20
nw16	1181590.00	3.84	1181590.00	9.41	1181590.00	7.89
nw05	132878.00	9.92	132878.00	4.59	132878.00	5.42
us01	9958.51	11.70	9963.07	14.19	9963.07	259.66
Average	48443.38	1.19	48653.81	0.87	48653.81	5.04

3.6 Concluding remarks

In this chapter we examined several concepts and techniques considering lower bound calculation for set partitioning problems. We focussed on Lagrangian relaxation and subgradient search techniques which do not need to use external linear programming solvers. We introduced several new subgradient search methods, the static convergent series method, the dynamic convergent series method and the bundle convergent series method, that are all based upon the convergent series method of Goffin (1977).

The convergent series method is applied rarely in practice, while the methods we examined work very well for the set partitioning problem. We have made several successful adjustment to this method to improve its performance: considering only part of the set partitioning tableau, using a starting solution and adjusting the parameters during the subgradient search. Moreover we have performed extensive testing to find good parameters to use these methods for the set partitioning problem.

The dynamic convergent series method we implemented outperforms other well-known techniques such as the classic subgradient search method of Held, Wolfe and Crowder (1974) and the volume algorithm of Barahona and Anbil (2000, 2002). Moreover, the results of the methods implemented in LaRSS allow us to find lower bounds close to and sometimes even better than the LP lower bounds, while the computing times are comparable.

Chapter 4

Upper bounds

This chapter briefly examines the literature on heuristics for set partitioning problems and describes our greedy heuristic to find upper bounds for set partitioning problems.

4.1 Literature on heuristics

Since the set partitioning problem is NP-complete, finding optimal solutions becomes increasingly difficult when the size of problem increases. Moreover, in real-life applications, time can be an important issue and finding a good solution fast with the use of heuristics is often more interesting than searching for the optimal solution. Therefore, in literature, much attention has been given to powerful heuristics to find very good, near-optimal solutions to the set partitioning problem. Especially in the older literature, heuristics receive more attention than optimization algorithms, since in that time computers could only solve very small set partitioning problems to optimality. Nowadays, computers are able to solve integer linear programming problems with over a million variables in less than one minute.

We mention some examples of these stand-alone heuristics in literature. Ryan and Falkner (1988) attempt to find a good solution to the set partitioning problem by imposing additional structure to the problem that is derived from real-life applications. This method appears to be effective in finding a good feasible solution rapidly. Atamtürk et al. (1995) describe a combined Lagrangian, linear programming and implication heuristic to generate provably good solutions. They also use preprocessing and probing techniques to speed up the algorithm. Their results show that the algorithm performs well in finding good, and often even optimal, solutions quickly. Wedelin (1995) describes a 0-1 optimizer that is very powerful in finding good solutions for crew scheduling and set covering problems fast. The optimizer can also be used to find solutions to the set partitioning problems, however results are not

presented. An example of a genetic algorithm for the set partitioning problem can be found in Chu and Beasley (1998). They report on good results as well, finding optimal or near-optimal solutions very quickly for all problems in their test set.

The aim of the heuristic described in this chapter is to find upper bounds, or primal solutions, to the set partitioning problem very quickly to support the branch and bound procedure. These upper bounds can be used to remove columns and to speed up the optimization process. This in contrast to the heuristics described above, where the aim is to find very good near-optimal solutions. The heuristic described here is a very simple greedy heuristic based on Fleuren (1988). An example of a comparable greedy heuristic can be found in Hoffman and Padberg (1993).

4.2 Primal heuristic

4.2.1 Implementation

The primal heuristic used in LaRSS is based on the greedy primal heuristic of Fleuren (1988). In the heuristic we consider three row orderings, discussed in Section 4.1.2, and perform an iterative procedure. In every iteration, we consider the rows in the given sequence and add the column with the lowest reduced costs that covers the next row to the partial solution. The iterations end if either the problem becomes infeasible or we find a feasible solution. In the first case, the first row in the ordering that cannot be covered is put in front of the sequence and the next iteration is started. In the second case, the middle row in the row ordering is put in front and the next iteration is started. If the primal heuristic does not find a solution to the problem, the upper bound is infinity. Figure 4.1 gives a schematic overview of the primal heuristic.

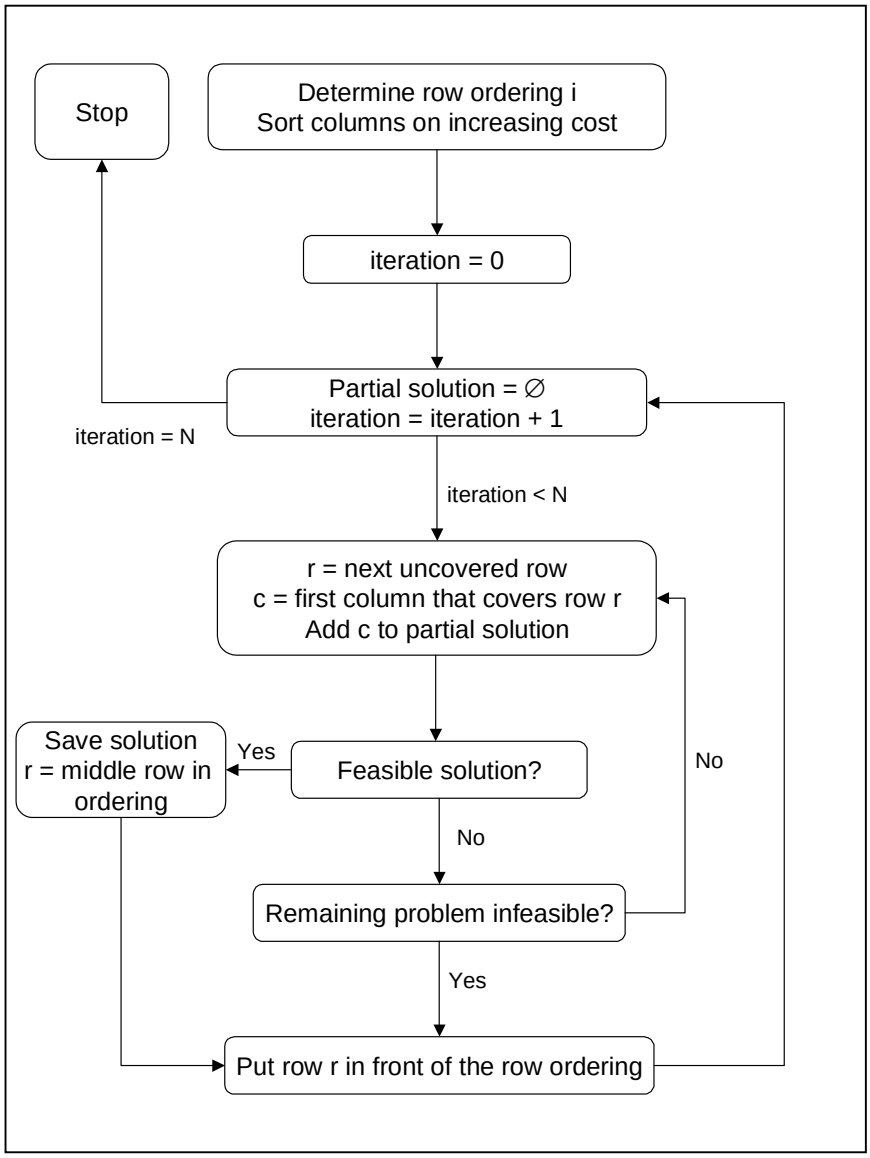
4.2.2 Row ordering

The performance of the primal heuristic obviously depends on the ordering of the rows. We consider the following three row orderings:

1. The rows are sorted on decreasing dual values u_r. This row ordering is based on the perception that rows with a high dual value have great influence on the objective value of the problem and are thus considered first.
2. The rows are ordered on increasing number of non-zeros. This row ordering is based upon the idea that rows with a small number of non-zeros are more difficult to cover and thus can be best considered in the beginning of the heuristic.
3. The rows are ordered on cover frequency. The cover frequency of row r with row s, cf(r,s) is the number columns in J(r) that also cover row s and can be seen as a measure for the overlap between rows r and s. This row ordering is determined as follows:

a. order[0] = first row of problem in original sequence
   $R' = \{order[0]\}$
   i = 1
b. order[i] = $\underset{s \in R \setminus R'}{\text{argmax}} \; cf(order[i-1], s)$
   $R' = R' \cup \{order[i]\}$
   i = i + 1
c. If R' = R then stop, else go to b.

Figure 4.1: The implementation of the primal heuristic



We now discern two strategies. In the first strategy, we take one of the row orderings described before and perform N iterations. In the second strategy, we use all three of the row orderings and reorder the rows after N/3 iterations. Computational results for these strategies are discussed in Sections 4.2.1 and 4.2.2, respectively.

Table 4.1: Computational results of the primal heuristic with 600 iterations for the three different row orderings

		Row order 1			Row order 2			Row order 3		
Name	Optimum	UB	Δ Columns	Time	UB	Δ Columns	Time	UB	Δ Columns	Time
nw41	11307	11457	82%	0.02	11457	82%	0.00	11307	83%	0.00
nw32	14877	14877	77%	0.00	14877	77%	0.00	14877	77%	0.00
nw40	10809	10896	78%	0.00	10896	78%	0.02	10848	80%	0.00
nw08	35894	35894	79%	0.00	35894	79%	0.00	35894	79%	0.00
nw15	67743	67743	85%	0.00	67743	85%	0.00	67743	85%	0.00
nw21	7408	7850	67%	0.00	7408	71%	0.00	7528	69%	0.00
nw22	6984	6984	82%	0.00	7144	79%	0.00	6984	82%	0.00
nw12	14118	14118	70%	0.00	14118	70%	0.00	14118	70%	0.00
nw39	10080	10758	78%	0.00	10758	78%	0.00	10758	78%	0.00
nw20	16812	17157	68%	0.00	17634	60%	0.00	17157	68%	0.00
nw23	12534	13904	27%	0.00	13988	26%	0.02	14064	26%	0.00
nw37	10068	11286	71%	0.00	10233	81%	0.00	10233	81%	0.00
nw26	6796	6942	56%	0.00	6942	56%	0.00	6942	56%	0.02
nw10	68271	68271	76%	0.00	68271	76%	0.00	68271	76%	0.00
nw34	10488	10488	79%	0.00	10701	79%	0.00	10488	79%	0.00
Heart	180	INF	0%	0.02	INF	0%	0.00	INF	0%	0.00
nw43	8904	8974	89%	0.00	8904	91%	0.00	8974	89%	0.00
nw42	7656	7684	71%	0.00	7684	71%	0.00	7714	71%	0.00
Delta	126	INF	0%	0.00	INF	0%	0.02	INF	0%	0.00
nw28	8298	8688	45%	0.00	8688	45%	0.00	8688	45%	0.00
nw25	5960	6286	65%	0.00	7526	48%	0.00	6286	65%	0.00
nw38	5558	5558	59%	0.00	5558	59%	0.02	5558	59%	0.00
nw27	9933	9933	60%	0.00	9933	60%	0.00	9933	60%	0.00
nw24	6314	6568	63%	0.00	6568	63%	0.00	6568	63%	0.00
nw35	7216	7896	63%	0.00	7896	63%	0.00	7216	69%	0.02
nw36	7314	7328	66%	0.00	7328	66%	0.00	7502	59%	0.00
Snowflake	34	INF	0%	0.06	INF	0%	0.05	INF	0%	0.06
Fives	12	INF	0%	0.00	INF	0%	0.02	INF	0%	0.02
Meteor	60	INF	0%	0.00	INF	0%	0.02	INF	0%	0.00
nw29	4274	4432	75%	0.00	4802	61%	0.02	4430	75%	0.00
nw30	3942	4294	68%	0.00	4294	68%	0.00	4294	68%	0.00
nw31	8038	8046	64%	0.00	8144	64%	0.00	8046	64%	0.00
nw19	10898	10898	74%	0.00	10898	74%	0.00	10898	74%	0.00
nw33	6678	7536	67%	0.00	8812	37%	0.00	6682	75%	0.02
nw09	67760	67760	74%	0.00	67760	74%	0.00	67760	74%	0.00
nw07	5476	5476	60%	0.00	5476	60%	0.00	5476	60%	0.00
aa02	30494	30494	73%	0.00	30494	73%	0.00	30494	73%	0.02
nw06	7810	10438	47%	0.03	9686	64%	0.02	9004	77%	0.03
aa04	26374	30740	0%	0.02	INF	0%	0.02	INF	0%	0.02
aa06	27040	27158	44%	0.03	27780	0%	0.05	27249	26%	0.03
kl01	1086	1086	77%	0.02	1091	69%	0.02	1088	74%	0.02
aa05	53839	53949	56%	0.03	54180	36%	0.08	53949	56%	0.05
aa03	49649	50228	22%	0.05	49684	71%	0.06	49680	71%	0.05
nw11	116256	119943	44%	0.02	117333	56%	0.00	116265	64%	0.00
aa01	56137	INF	0%	0.02	INF	0%	0.03	INF	0%	0.03
nw18	340160	364330	0%	0.08	372952	0%	0.06	364060	0%	0.08
us02	5965	5965	42%	0.00	5965	42%	0.02	5965	42%	0.02
nw13	50146	50526	66%	0.02	51302	52%	0.00	50276	67%	0.00
us04	17854	17854	18%	0.02	17862	18%	0.02	17854	18%	0.00
kl02	219	219	37%	0.03	219	37%	0.03	219	37%	0.05
nw03	24492	25464	87%	0.06	25182	88%	0.06	25464	87%	0.06
nw01	114852	115908	95%	0.20	118221	85%	0.16	116925	92%	0.41
us03	5338	5807	23%	0.08	5914	22%	0.09	5813	23%	0.08
nw04	16862	22494	0%	0.99	22494	0%	0.94	26126	0%	0.81
nw02	105444	105444	97%	0.16	105444	97%	0.16	105444	97%	0.14
nw17	11115	12600	58%	0.38	12438	60%	0.55	11913	64%	0.19
nw14	61844	61844	77%	0.14	61844	77%	0.14	61844	77%	0.16
nw16	1181590	1184212	93%	0.34	1184212	93%	0.36	1183598	93%	0.36
nw05	132878	132878	70%	0.34	132878	70%	0.34	132878	70%	0.34
us01	10036	10081	32%	2.50	10472	31%	1.73	10149	32%	2.81
Average	48772	54179	55%	0.09	54830	54%	0.08	54519	57%	0.10

4.3 Computational results

4.3.1 Fixed row ordering

Table 4.1 shows the results of the primal heuristic with 600 iterations for the three different row orderings specified in Section 4.1.2. For each of these row orderings, the table shows the upper bound found, the decrease in columns after reduced cost fixing and the computing time of the primal heuristic. Since two out of the three row orderings that are considered require the knowledge of a dual feasible vector λ, we examine the results of the primal heuristic in combination with a subgradient search algorithm. The results reported in Table 4.1 are obtained by first performing preprocessing, i.e. the equal columns, contained rows, clique and equal rows techniques, then the dynamic convergent series method ($\alpha_1 = 0.95$, $\alpha_2 = 0.975$, $\alpha_3 = 0.99875$, $C^0 = 1{,}000$ and $Q = 25$) and finally employing the primal heuristic. The times reported are the computing times of the primal heuristic.

Table 4.2 summarizes the results of Table 4.1 and gives some statistics for each of the three row orderings. Note that the instances for which no bound is found during the primal heuristic are left out of the calculation of the maximum and average deviation from the optimum. Row ordering 3 seems to be the best of the three orderings, since the average deviation is the lowest, the optimal solution is found for 21 instances and this row ordering gives, of all three row orderings, the best upper bound for 47 instances. The total time of row ordering 3 is also the highest, although the differences are rather small.

Table 4.2: Summary of the results of the primal heuristic with N = 600

	Row order 1	**Row order 2**	**Row order 3**
# Times no bound found	6	7	7
# Times best of three orderings	44	34	47
# Times optimal	20	18	21
Total time (s)	5.63	5.08	5.86
Maximal deviation from optimum	34%	33%	55%
Average deviation from optimum	4.01%	4.53%	3.01%

4.3.2 Variable row ordering

This section examines the results of the primal heuristic when we perform N/3 iterations for each of the row orderings consecutively. Tables 4.3 and 4.4 show these results for our test set with N = 300, N = 600 and N = 900.

Table 4.3: Computational results of the primal heuristic with variable row ordering for different values of N

		N = 300			N = 600			N = 900		
Name	Optimum	UB	Δ Columns	Time	UB	Δ Columns	Time	UB	Δ Columns	Time
nw41	11307	11307	83%	0.00	11307	83%	0.00	11307	83%	0.00
nw32	14877	14877	77%	0.00	14877	77%	0.00	14877	77%	0.00
nw40	10809	10848	80%	0.00	10848	80%	0.00	10848	80%	0.00
nw08	35894	35894	79%	0.00	35894	79%	0.00	35894	79%	0.00
nw15	67743	67743	85%	0.00	67743	85%	0.00	67743	85%	0.00
nw21	7408	7408	71%	0.00	7408	71%	0.00	7408	71%	0.00
nw22	6984	6984	82%	0.00	6984	82%	0.00	6984	82%	0.00
nw12	14118	14118	70%	0.00	14118	70%	0.00	14118	70%	0.00
nw39	10080	10758	78%	0.00	10758	78%	0.00	10758	78%	0.00
nw20	16812	17157	68%	0.00	17157	68%	0.00	17157	68%	0.00
nw23	12534	13904	27%	0.00	13904	27%	0.00	13904	27%	0.00
nw37	10068	10233	81%	0.00	10233	81%	0.00	10233	81%	0.00
nw26	6796	6942	56%	0.00	6942	56%	0.02	6942	56%	0.00
nw10	68271	68271	76%	0.00	68271	76%	0.00	68271	76%	0.00
nw34	10488	10488	79%	0.00	10488	79%	0.00	10488	79%	0.00
Heart	180	INF	0%	0.00	INF	0%	0.02	INF	0%	0.00
nw43	8904	8904	91%	0.00	8904	91%	0.00	8904	91%	0.00
nw42	7656	7684	71%	0.00	7684	71%	0.00	7684	71%	0.00
Delta	126	INF	0%	0.00	INF	0%	0.02	INF	0%	0.00
nw28	8298	8688	45%	0.00	8688	45%	0.00	8688	45%	0.00
nw25	5960	6286	65%	0.00	6286	65%	0.00	6286	65%	0.00
nw38	5558	5558	59%	0.00	5558	59%	0.00	5558	59%	0.00
nw27	9933	9933	60%	0.00	9933	60%	0.00	9933	60%	0.00
nw24	6314	6568	63%	0.00	6568	63%	0.00	6568	63%	0.00
nw35	7216	7216	69%	0.00	7216	69%	0.00	7216	69%	0.00
nw36	7314	7328	66%	0.00	7328	66%	0.00	7328	66%	0.00
Snowflake	34	INF	0%	0.05	INF	0%	0.06	INF	0%	0.06
Fives	12	INF	0%	0.00	INF	0%	0.00	INF	0%	0.00
Meteor	60	INF	0%	0.00	INF	0%	0.00	INF	0%	0.00
nw29	4274	4430	75%	0.00	4430	75%	0.00	4430	75%	0.00
nw30	3942	4294	68%	0.00	4294	68%	0.00	4294	68%	0.02
nw31	8038	8046	64%	0.00	8046	64%	0.00	8046	64%	0.02
nw19	10898	10898	74%	0.00	10898	74%	0.00	10898	74%	0.00
nw33	6678	6682	75%	0.00	6682	75%	0.00	6682	75%	0.02
nw09	67760	67760	74%	0.00	67760	74%	0.00	67760	74%	0.00
nw07	5476	5476	60%	0.00	5476	60%	0.00	5476	60%	0.00
aa02	30494	30494	73%	0.00	30494	73%	0.00	30494	73%	0.00
nw06	7810	9004	77%	0.00	9004	77%	0.02	9004	77%	0.03
aa04	26374	INF	0%	0.02	INF	0%	0.02	INF	0%	0.02
aa06	27040	27525	3%	0.02	27380	10%	0.05	27249	26%	0.06
kl01	1086	1088	74%	0.02	1088	74%	0.02	1086	77%	0.02
aa05	53839	54096	42%	0.03	53949	56%	0.05	53949	56%	0.09
aa03	49649	49684	71%	0.03	49684	71%	0.05	49680	71%	0.06
nw11	116256	116265	64%	0.00	116265	64%	0.02	116265	64%	0.00
aa01	56137	INF	0%	0.02	INF	0%	0.03	INF	0%	0.03
nw18	340160	364060	0%	0.03	364060	0%	0.06	364060	0%	0.11
us02	5965	5965	42%	0.02	5965	42%	0.02	5965	42%	0.02
nw13	50146	50276	67%	0.02	50276	67%	0.02	50276	67%	0.02
us04	17854	17854	18%	0.00	17854	18%	0.02	17854	18%	0.02
kl02	219	219	37%	0.03	219	37%	0.03	219	37%	0.05
nw03	24492	25182	88%	0.06	25182	88%	0.06	25182	88%	0.06
nw01	114852	115908	95%	0.16	115908	95%	0.22	115908	95%	0.31
us03	5338	5807	23%	0.08	5807	23%	0.09	5807	23%	0.13
nw04	16862	22494	0%	0.50	22494	0%	0.92	22494	0%	1.38
nw02	105444	105444	97%	0.14	105444	97%	0.16	105444	97%	0.14
nw17	11115	11913	64%	0.27	11913	64%	0.38	11913	64%	0.49
nw14	61844	61844	77%	0.16	61844	77%	0.14	61844	77%	0.16
nw16	1181590	1183598	93%	0.33	1183598	93%	0.36	1183598	93%	0.36
nw05	132878	132878	70%	0.36	132878	70%	0.34	132878	70%	0.36
us01	10036	10149	32%	1.92	10081	32%	2.38	10081	32%	2.81
Average	48772	54423	56%	0.07	54416	57%	0.09	54414	57%	0.11

Table 4.4: Summary of the results of the primal heuristic with variable row ordering

	N = 300	**N = 600**	**N = 900**
# Times no bound found	7	7	7
# Times best of three values of N	55	57	60
# Times optimal	23	23	24
Total time (s)	4.27	5.58	6.85
Maximal deviation from optimum	33%	33%	33%
Average deviation from optimum	2.47%	2.44%	2.43%

Obviously, the quality of the bound found by this heuristic, as well as the computing time, increases with the value of N. From the second row of the table we can deduce that for 55 instances, the bound found is the same for N = 300, N = 600 and N = 900. Moreover, when N = 600, better bounds are found for two more instances, while the heuristic always finds the best bound when N = 900. For N = 900, the heuristic finds the optimal solution for 24 out of the 60 instances and the average deviation is 2.43%. For all three values of N, the average deviation is lower than in the static row orderings cases. The variable row ordering strategy seems to work better. Altogether, the results of the six different strategies considered do not differ much in computing time or the upper bound found and the performance of the primal heuristic is quite robust.

4.4 Concluding remarks

This chapter discussed a primal heuristic that can be used to calculate upper bounds for set partitioning problems. Since the value of the upper bounds found is of limited importance for the performance of LaRSS, the speed of the heuristic is more important than the quality of the bounds and a very simple greedy heuristic is used. The greedy heuristic developed is able to find good upper bounds rapidly. Some sensitivity analysis is presented on the ordering of the rows and the number of iterations. The performance of the heuristic does not vary much when these characteristics are changed. The version of the heuristic that is implemented in LaRSS uses all three of the row orderings consecutively and performs 200 iterations for each row ordering.

Chapter 5

Branch and bound

This chapter discusses several branch and bound techniques for set partitioning problems. Some insight is provided in the implementation of a branch and bound procedure and several static as well as dynamic branching rules are described and compared to each other via computational experiments. Moreover, attention is given to the use of dual heuristics and Lagrangian relaxation during branch and bound.

5.1 Introduction

Branch and bound methods find the solution to an integer program by efficiently and intelligently enumerating the points in the feasible region of the problem. For a detailed discussion of branch and bound methods, see Winston (1994) and Papadimitriou and Steiglitz (1982). Section 5.1.1 briefly considers the literature on branch and bound methods for set partitioning problems.

A branch and bound algorithm searches the total solution space, i.e. all vectors $\{x_j\}_{j \in J}$, $x_j \in \{0,1\}$ in a systematic way to find the best solution. The search consists of constructing partial solutions by assigning the value of 0 or 1 to some variables and investigating the implications of these assignments. In every node of the branch and bound tree we either add a column to the partial solution, i.e. fix a variable x_j to 1, or we fathom the node, meaning that we do not extend the partial solution anymore. Section 5.1.2 discusses the three fathoming criteria. Sections 5.2 – 5.4 examine several branching strategies.

5.1.1 Literature

With the advances in linear programming software in the last decades, the attention

for branch and cut methods and linear programming based branch and bound algorithms has received more and more attention in the literature. In the classical linear programming based branch and bound or branch and cut algorithm, the branching is done on the values of the primal solution to the linear programming problem, creating two branched for every fractional solution value. In this way, a binary tree is created as discussed in Section 5.2. This chapter does not consider branch and cut or linear programming based branching methods. More information on branch and cut methods for set partitioning problems can be found in for example Balas and Padberg (1976) and Hoffman and Padberg (1993). Borndörfer (1998) gives an extensive discussion and several computational results on branch and cut methods as well as linear programming based branch and bound.

Relatively little information can be found in the literature on general branch and bound algorithms that can be applied without the use of linear programming software. Examples of implementations of branch and bound algorithms for set partitioning can be found in Marsten (1974), Balas and Padberg (1976), Albers (1980), Fisher and Kedia (1986) and Fleuren (1988). The dynamic constraint-based branching method that is applied in LaRSS has not been described in literature before.

5.1.2 Fathoming

In every node of the branch and bound tree, we have a partial solution to the set partitioning problem and a corresponding induced subproblem, as discussed in Section 3.1.3. We say that an induced subproblem is fathomed if the corresponding partial solution is a feasible solution to the set partitioning problem or we can show that it can never be extended to a better feasible solution than the best known solution at that point. Given a partial solution x^p, we can thus define three fathoming criteria:

1. The vector x^p fulfills the constraints [1.2] and [1.3] and thus forms a feasible solution to the set partitioning problem.
2. The induced subproblem defined by partial solution x^p cannot contain a solution and x^p can never be extended to a feasible solution, since at least one row in the induced subproblem cannot be covered.
3. Given a dual feasible vector u^f and an upper bound to the problem, the lower bound on the induced subproblem plus the costs of the partial solution exceed the upper bound UB:

$$\sum_{j \in J} cr_j \cdot x_j + \sum_{r \in R} u_r^f > UB \qquad [5.1]$$

5.1.3 Introduction to this chapter

This chapter considers three different branching strategies: classical variable-based

branching, static constraint-based branching and dynamic constraint-based branching. Sections 5.2 – 5.4 discuss the theory, implementation issues and computational results considering these branching strategies.

Table 5.1: Problem characteristics before branch and bound

Name	# Rows	# Columns	LB	UB	Optimum	Solved?
nw41	3	5	11307.00	11307	11307	ves
nw32	11	18	14570.00	14877	14877	no
nw40	8	11	10658.09	10809	10809	no
nw08	19	22	35894.00	35894	35894	yes
nw15	31	407	67743.00	67743	67743	yes
nw21	4	4	7408.00	7408	7408	yes
nw22	5	5	6984.00	6984	6984	yes
nw12	11	11	14118.00	14118	14118	yes
nw39	6	16	9868.50	10410	10080	no
nw20	20	47	16624.72	16965	16812	no
nw23	14	37	12317.00	12534	12534	no
nw37	3	3	10068.00	10068	10068	yes
nw26	3	3	6796.00	6796	6796	yes
nw10	20	12	68271.00	68271	68271	yes
nw34	7	7	10453.50	10488	10488	no
Heart	134	855	179.54	INF	180	no
nw43	4	4	8904.00	8904	8904	yes
nw42	16	29	7484.98	7684	7656	no
Delta	111	989	126.00	INF	126	no
nw28	2	2	8298.00	8298	8298	yes
nw25	19	48	5852.00	6286	5960	no
nw38	12	23	5550.87	5688	5558	no
nw27	4	4	9933.00	9933	9933	yes
nw24	1	15	6314.00	6314	6314	yes
nw35	14	59	7206.00	7896	7216	no
nw36	13	72	7259.96	7328	7314	no
Snowflake	170	1996	11.96	INF	34	no
Fives	72	2440	11.99	INF	12	no
Meteor	60	1370	60.00	INF	60	no
nw29	15	52	4189.80	4344	4274	no
nw30	21	59	3723.43	4294	3942	no
nw31	26	24	7980.00	8046	8038	no
nw19	5	5	10898.00	10898	10898	yes
nw33	4	4	6678.00	6678	6678	yes
nw09	33	204	67760.00	67760	67760	yes
nw07	36	3108	5476.00	5476	5476	yes
aa02	302	3958	30494.00	30494	30494	yes
nw06	37	698	7639.78	8984	7810	no
aa04	296	6289	25870.36	29833	26374	no
aa06	392	2357	26973.26	27155	27040	no
kl01	40	179	1083.61	1086	1086	no
aa05	397	1513	53721.42	53949	53839	no
aa03	416	419	49607.10	49649	49649	no
nw11	29	1544	112403.86	116256	116256	no
aa01	498	7659	55519.00	INF	56137	no
nw18	77	6250	328735.44	342998	340160	no
us02	21	10	5965.00	5965	5965	yes
nw13	49	94	50131.31	50206	50146	no
us04	53	79	17729.56	17854	17854	no
kl02	57	2298	215.05	219	219	no
nw03	53	197	24447.00	24759	24492	no
nw01	132	50113	114852.00	114852	114852	yes
us03	24	6	5338.00	5338	5338	yes
nw04	35	7574	16310.18	17264	16862	no
nw02	143	1194	105444.00	105444	105444	yes
nw17	54	512	10874.03	11382	11115	no
nw14	69	10474	61844.00	61844	61844	yes
nw16	62	64	1181590.00	1181590	1181590	yes
nw05	61	56	132878.00	132878	132878	yes
us01	86	392	9958.51	10056	10036	no

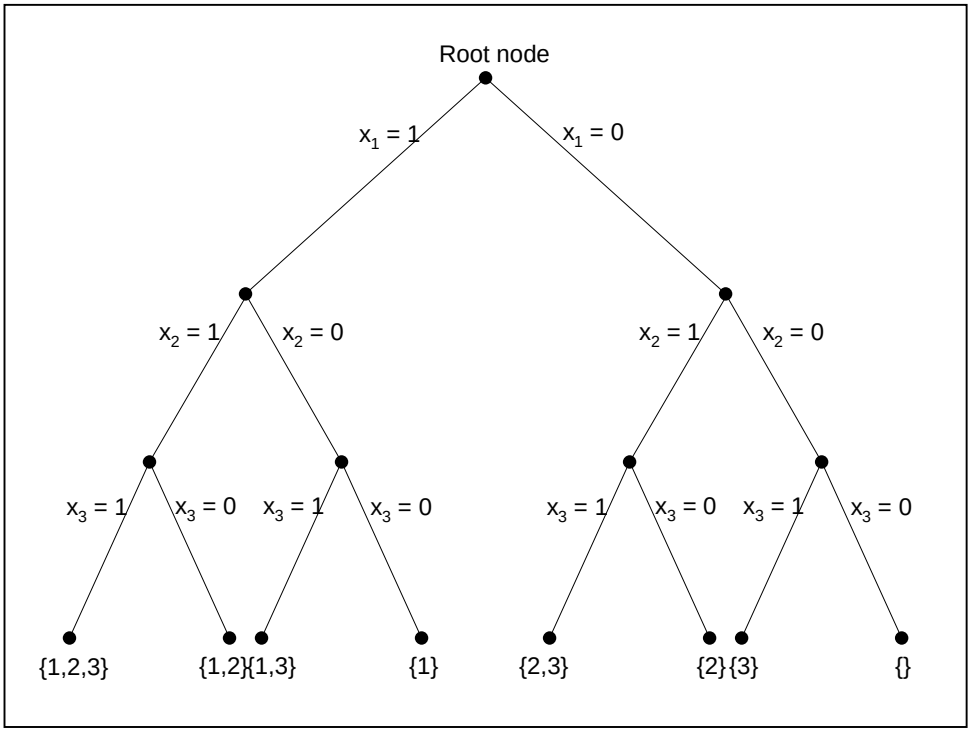
The computational results discussed in these sections are the results of applying branch and bound on the preprocessed test set. In this case “preprocessed” refers not only to the problem reduction techniques, but also to the use of the subgradient search algorithm and the dual- and primal heuristics. The sequence in which the techniques are applied within LaRSS is discussed in Chapter 7.

Table 5.1 shows the characteristics of the test set before branch and bound, including the number of rows and columns, the lower- and upper bounds and the optimal value. Note that for 25 out of the 60 problems in our test set, the optimal solution is found by the primal heuristic and optimality can be concluded by comparing the lower- and upper bounds. These problems do not require the branch and bound procedure and they are excluded from the computational experiments in this chapter. This is indicated in the last column of Table 5.1.

5.2 Classical variable-based branching

The classical branching strategy is variable-based, meaning that we consider the variables in a given order and branch on value of the variables. Otherwise said: we branch on the decision whether or not to take the column belonging to that variable in the partial solution. Variable-based branching results in a binary search tree, as illustrated by Figure 5.1.

Figure 5.1: Example of a branching tree for variable-based branching



When the solution to the linear programming relaxation problem is known, the value of the primal solution variables can be used for determining the sequence in which the variables are considered in the branching tree. Since we do not have such information at our disposal, we use a static sequence to branch the variables, or columns. We consider three column orderings:

- Sort columns on increasing reduced costs (cr_j)
- Sort columns on the number of nonzero elements (|R(j)|)
- Take the original column ordering

We implement the depth-first search of this binary search tree. This means that we examine the tree from the left to the right, going down the tree as far as possible until we can fathom a node. In this case we go back one step and go down the tree again. In every node we check the feasibility of the induced subproblem. Figure 5.2 gives a brief outline of such a depth-first search strategy. Section 5.4 discusses the computational results of this branching strategy for all three column orderings.

Figure 5.2: Outline of the variable-based branching strategy

```
Function BranchAndBound

CO = ordered set of all columns
FOR(column in CO)
        Branch(column)
ENDFOR


Function Branch(column)

Add column to the partial solution
IF(feasible solution)
    save solution
ELSE
    IF(node not fathomed)
        FOR(next_column in CO, next_column > column)
            IF(next_column not blocked by partial solution)
                Branch(next_column)
            ENDIF
        ENDFOR
    ENDIF
ENDIF
Remove column from partial solution
```

5.3 Constraint-based branching

For every row r in the set partitioning tableau, we have to choose exactly one column in J(r) in the solution. We can thus branch on the decision which column to choose to cover a particular row. Consider a small example with three rows; r, s and t, with

J(r) = {1,2,3,4}, J(s) = {5,6} and J(t) = {7,8,9}. Figure 5.3 shows the branching tree if a complete search for these three rows is performed.

Again, we consider a depth-first search of the search tree. Figure 5.4 gives a brief outline of the constraint-based branching algorithm. The orderings of rows as well as the ordering of the columns obviously influence the performance of this algorithm. We consider two types of row orderings: static and dynamic, which will be discussed in Sections 5.3.1 and 5.3.2, respectively. In all these strategies we check the feasibility of the induced subproblem in every node. The columns are ordered on increasing reduced costs, since this offers possibilities for reducing the computing time of the branch and bound procedure. To see this, suppose that we are at a certain node in the branching tree, where a partial solution x^p and an induced subproblem with row set R_2 and column set J_2 are known. Furthermore suppose that row r is chosen to be covered next. For every column $j \in J(r)$ we can now conclude that it is only profitable to add this column to the partial solution if:

$$cr_j \leq UB - \left(\sum_{k \in J} cr_k \cdot x_k^p + \sum_{r \in R} u_r^f \right) \qquad [5.2]$$

Since the columns are sorted on increasing reduced costs, we can stop searching for this row when we arrive at the first column $j \in J(r)$ for which

$$cr_j > UB - \left(\sum_{k \in J} cr_k \cdot x_k^p + \sum_{r \in R} u_r^f \right) \qquad [5.3]$$

Figure 5.3: Example of a branching tree for constraint-based branching

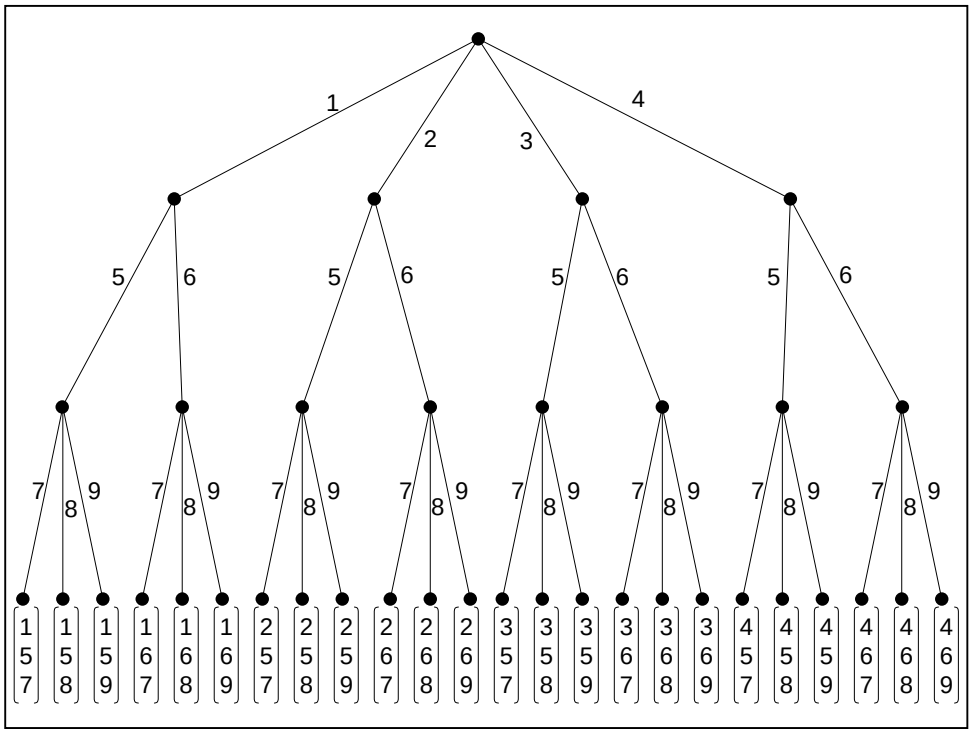


Figure 5.4: Outline of the constraint-based branching strategy

```
Function BranchAndBound

Order the set of columns on increasing reduced costs
row = choose the next row
Branch(row)

Function Branch(row)

FOR(column in J(row))
        IF(column not blocked by partial solution)
                Add column to partial solution
                IF(feasible solution)
                        save solution
                ELSE
                        IF(node not fathomed)
                                next_row = choose the next row
                                branch(next_row)
                        ENDIF
                ENDIF
                Remove column from partial solution
        ENDIF
ENDFOR
```

5.3.1 Static constraint-based branching

In case of static constraint-based branching, we determine the ordering of the rows before the branch and bound procedure starts and do not change this sequence during the branching process. The static row sequences we consider are the same as those used for the primal heuristic:

- Sort the rows by decreasing value of the dual value u_r
- Sort the rows by increasing number of nonzero elements, $|J(r)|$
- Sort the rows by cover frequency. The cover frequency of row r with row s, cf(r,s) is the number columns in J(r) that also cover row s and can be seen as a measure for the overlap between rows r and s. This row sequence is determined beforehand with the procedure discussed in Section 4.1.2

Compared to variable-based branching, we now use the extra information that for every row r, only one column in R(j) will be chosen in the solution. Section 5.4 discusses the computational results of applying static constraint-based branching with these sequences on our test set.

5.3.2 Dynamic constraint-based branching

In the dynamic constraint-based branching strategy, we do not determine the row sequence before the branching procedure, but we choose the next row to cover

during the procedure. The advantage of this adjustment is that we can incorporate information that comes available during the branch and bound, like the number of active columns that cover a particular row. We say that a column j is active if it is not blocked by the partial solution and the reduced costs of the column, cr_j, fulfill requirement [5.2]. We consider two dynamic strategies:

- In every level of the branching tree we choose the row that is covered by the smallest number of active columns.
- In every level of the branching tree we choose the row that has the largest overlap with the row in the previous level.

Compared to static constraint-based branching, we now use extra information about active columns and the row sequence that becomes available during the branching process. Computational results for these two dynamic strategies are discussed in Section 5.4.

5.4 Computational results

This section discusses the results of applying the several branching strategies on our test set. The problems are preprocessed, as discussed in Section 5.1.2. In every experiment, the branching procedure is stopped when the time needed to perform the branching exceeds ten minutes.

5.4.1 Variable-based branching

Table 5.2 shows the results of applying variable-based branching on our test set for three different column orderings. When columns are sorted on reduced costs cr_j, there are 11 problems for which the branching process does not finish in 10 minutes and the total time exceeds 133 minutes. When the columns are sorted on increasing number of nonzero elements, the branching does not finish within 10 minutes for 13 out of the 35 problems and the total time is over 144 minutes. Finally, when the original column sequence is used, there are 12 problems for which the branching cannot conclude optimality within 10 minutes and the total time is at least 128 minutes. The performance of all three strategies is very poor, since the method used is not very sophisticated and does not make use of the structure of the problem.

Table 5.2: Results of variable-based branching for three different column orderings

	Sort columns on cr_j		Sort columns on |R(j)|		Original column sequence	
Name	**Time (s)**	**Nodes**	**Time (s)**	**Nodes**	**Time (s)**	**Nodes**
nw32	0.02	88	0.03	88	0.03	88
nw40	0.02	8	0.16	9	0.08	9
nw39	0.03	16	0.13	46	0.06	12
nw20	0.05	212	0.16	206	0.09	221
nw23	0.03	204	0.14	206	0.03	258
nw34	0.03	16	0.13	16	0.02	13
Heart	> 600.00	> 112000000	> 600.00	> 120000000	> 600.00	> 138000000
nw42	0.08	92	0.08	83	0.08	86
Delta	> 600.00	> 90000000	> 600.00	> 90000000	> 600.00	> 90000000
nw25	0.06	108	0.06	209	0.06	155
nw38	0.03	20	0.05	24	0.03	22
nw35	0.03	14	0.05	105	0.05	32
nw36	0.11	548	0.11	724	0.11	585
Snowflake	> 600.00	> 28000000	> 600.00	> 60000000	> 600.00	> 26000000
Fives	249.34	19598595	> 600.00	> 64000000	62.92	5220818
Meteor	52.73	5591708	42.27	5591708	81.94	11013112
nw29	0.09	169	0.19	200	0.11	173
nw30	0.08	241	0.16	470	0.08	215
nw31	0.09	85	0.17	51	0.09	37
nw06	0.66	186959	0.78	53637	1.08	77979
aa04	> 600.00	> 54000000	> 600.00	> 24000000	> 600.00	> 62000000
aa06	> 600.00	> 162000000	> 600.00	> 72000000	> 600.00	> 76000000
kl01	17.66	36137503	27.64	14574784	13.81	7935214
aa05	> 600.00	> 194000000	> 600.00	> 62000000	> 600.00	> 62000000
aa03	> 600.00	> 136000000	> 600.00	> 190000000	> 600.00	> 86000000
nw11	> 600.00	> 562000000	> 600.00	> 24000000	> 600.00	> 102000000
aa01	> 600.00	> 48000000	> 600.00	> 50000000	> 600.00	> 50000000
nw18	> 600.00	> 64000000	> 600.00	> 16000000	> 600.00	> 30000000
nw13	424.00	875844769	555.50	701915000	197.00	285308655
us04	0.81	165646	0.88	94508	0.75	60998
kl02	> 600.00	> 432000000	> 600.00	> 28000000	> 600.00	> 34000000
nw03	4.22	1111203	3.59	564721	3.25	439800
nw04	438.25	152351136	> 600.00	> 4000000	> 600.00	> 6000000
nw17	71.36	18724827	77.81	8867091	46.92	5456549
us01	162.38	54747807	181.94	17592705	101.66	5829038
Total time	> 8022		> 8692		> 7710	
# Not solved	11		13		12	

5.4.2 Static constraint-based branching

Table 5.3 shows the results of applying static constraint-based branching on the problems in our test set. The three different row orderings as discussed in Section 5.3.1 are considered. When the rows are ordered according to their dual values, the branching does not finish within 10 minutes for 7 out of the 35 problems. The total time in this case exceeds 70 minutes. When the rows are ordered according to the number of nonzero elements, the branching algorithm cannot conclude optimality within 10 minutes for 7 instances, while the total time exceeds 70 minutes. Finally, when the rows are ordered according to the cover frequency, the total time is over 56 minutes and optimality cannot be guaranteed within 10 minutes for 5 out of the 35 instances. Since we do use the structure of the problem, the results are somewhat better than the results of variable-based branching.

Table 5.3: Results of static constraint-based branching for three row orderings

	Sort rows on λ_r		Sort rows on |J(r)|		Sort rows on cover frequency	
Name	**Time (s)**	**Nodes**	**Time (s)**	**Nodes**	**Time (s)**	**Nodes**
nw32	0.00	9	0.00	9	0.00	9
nw40	0.00	5	0.00	5	0.00	4
nw39	0.00	18	0.00	9	0.00	6
nw20	0.00	55	0.02	21	0.00	8
nw23	0.00	12	0.00	9	0.00	13
nw34	0.00	5	0.00	5	0.00	5
Heart	9.38	1828381	18.84	3511032	1.13	214693
nw42	0.00	20	0.00	25	0.00	16
Delta	0.03	3506	5.06	604217	0.08	9559
nw25	0.00	23	0.00	16	0.00	14
nw38	0.00	7	0.00	6	0.00	6
nw35	0.00	10	0.00	9	0.00	14
nw36	0.00	66	0.00	56	0.00	58
Snowflake	> 600.00	> 22000000	> 600.00	> 22000000	> 600.00	> 18000000
Fives	22.50	1754540	8.81	772061	1.81	142279
Meteor	0.00	925	0.17	18353	0.02	439
nw29	0.00	36	0.00	31	0.00	22
nw30	0.00	31	0.00	13	0.00	13
nw31	0.00	12	0.00	14	0.00	12
nw06	0.02	742	0.00	150	0.00	147
aa04	> 600.00	> 50000000	> 600.00	> 76000000	> 600.00	> 60000000
aa06	> 600.00	> 110000000	> 600.00	> 114000000	> 600.00	> 106000000
kl01	0.02	9114	0.00	2524	0.00	2687
aa05	> 600.00	> 126000000	> 600.00	> 100000000	284.00	51767972
aa03	0.50	85111	0.56	118149	0.03	4301
nw11	0.13	27103	1.13	503372	1.27	554434
aa01	> 600.00	> 60000000	> 600.00	> 42000000	> 600.00	> 40000000
nw18	> 600.00	> 52000000	> 600.00	> 76000000	> 600.00	> 54000000
nw13	0.00	301	0.02	95	0.00	213
us04	0.00	63	0.00	276	0.00	213
kl02	> 600.00	> 242000000	> 600.00	> 334000000	77.39	25482046
nw03	0.03	543	0.00	867	0.00	900
nw04	1.27	52784	1.49	45460	1.31	59543
nw17	0.05	450	0.00	996	0.02	682
us01	0.08	6169	0.05	2175	0.00	1741
Total time	> 4233		> 4236		> 3367	
# Not solved	7		7		5	

5.4.3 Dynamic constraint-based branching

Table 5.4 shows the results of applying dynamic constraint-based branching on our test set. When the next row to branch is chosen according to the largest overlap with the preceding row, the branching procedure does not end within 10 minutes for 7 problem instances. The total time exceeds 73 minutes. When we choose the strategy to branch on the row with the smallest number of active columns, the branching procedure does not finish within 10 minutes for 2 out of the 35 instances. The total branching time on the remaining 33 problems is 12 seconds. The two problems that are not solved within 10 minutes, aa01 and aa04, are not solved within 10 minutes with any of the branching strategies discussed thus far. We will consider these problems in more detail in Section 5.5. Dynamic constraint-based branching on the row with the smallest number of active columns is the best branching strategy discussed thus far. In this method we try to make use of the structure of the set

partitioning problem in a smart way, by using a constraint-based branching that is dynamically adjusted. However, the drawback of the chosen lower bound calculation method is that lower bound improvement during branch and bound is very difficult and time-consuming. Therefore, especially when the gap between lower- and upper bound is large, the branch and bound procedure needs an extensive search to find the optimal solution and guarantee optimality.

Table 5.4: Results of dynamic constraint-based branching for two row orderings

	Smallest # active columns		Largest overlap	
Name	**Time (s)**	**Nodes**	**Time (s)**	**Nodes**
nw32	0.00	8	0.00	9
nw40	0.00	3	0.00	3
nw39	0.00	6	0.00	6
nw20	0.00	11	0.00	8
nw23	0.00	12	0.00	9
nw34	0.00	4	0.00	4
Heart	0.03	2475	1.56	74561
nw42	0.00	11	0.00	16
Delta	0.02	614	0.72	33022
nw25	0.00	8	0.00	17
nw38	0.00	6	0.00	5
nw35	0.00	4	0.00	4
nw36	0.00	27	0.00	56
Snowflake	2.34	42734	> 600.00	> 6000000
Fives	0.02	188	189.05	4909638
Meteor	0.02	338	0.16	5996
nw29	0.00	12	0.00	29
nw30	0.00	21	0.00	27
nw31	0.00	10	0.00	7
nw06	0.00	129	0.02	996
aa04	> 600.00	> 32000000	> 600.00	> 10000000
aa06	2.94	283242	> 600.00	> 20000000
kl01	0.00	444	0.00	2355
aa05	2.70	368061	> 600.00	> 42000000
aa03	0.00	1960	0.39	37900
nw11	0.64	214073	8.50	714202
aa01	> 600.00	> 28000000	> 600.00	> 8000000
nw18	2.69	349927	> 600.00	> 12000000
nw13	0.00	72	0.00	53
us04	0.00	28	0.00	269
kl02	0.47	121928	> 600.00	> 40000000
nw03	0.00	45	0.02	1613
nw04	0.28	9267	14.00	71286
nw17	0.00	141	0.02	641
us01	0.00	342	0.05	1661
Total time	> 1212		> 4414	
# Not solved	2		7	

5.5 Enhancing the branch and bound procedure

Solution methods based on linear programming (LP) relaxation have two important advantages over Lagrangian relaxation-based methods. The first advantage is that, in a branch and bound procedure, lower bounds can be updated very quickly when the basis of the LP solution changes. The second advantage is that, using the linear programming information, cuts designed to improve the lower bound, can be

evaluated easily. In this section we will examine if and how we can use Lagrangian relaxation and dual heuristics during the branch and bound procedure. In Chapter 6 we will briefly consider the possible value of cuts in LaRSS. In our computational experiments in this section, we will focus mainly on the instances aa01 and aa04 from our test set, since these instances are not solved in reasonable time in the basic dynamic constraint-based branching method. Section 5.5.1 examines some characteristics of these two problems. Sections 5.5.2 – 5.5.4 examine the value of the dual update heuristic, the dual 3OPT heuristic and the subgradient search algorithm within the branch and bound procedure. To limit the amount of results reported, we do not discuss the research performed to determine the appropriate parameters for all these methods and consider these given.

5.5.1 Two difficult instances

The test set contains two problems that are difficult to solve relative to the other problems in the set: aa01 and aa04. The literature has paid much attention to these instances. Hoffman and Padberg (1993) report that these instances require significantly more computational effort than the rest. In their branch and cut approach, they developed extra software, especially for the constraint generator, to be able to handle these “outliers” satisfactorily. According to Hoffman and Padberg, there is nothing unique about these instances in terms of size, density, distribution of the nonzeros or distribution of the cost data, that makes them so difficult to solve. They report computing times of 4.0 hours for aa01 and 38.7 hours for aa04. In his dissertation, Borndörfer (1998) also mentions difficulty with solving problems aa01 and aa04. He finds that closing the gap from the dual side is what makes these problems difficult.

In the basic constraint-based branching method, these problems are not solved in two hours. Although there is no clear answer to the question what makes these problems more difficult to solve, in our research on aa01 we did discover some remarkable characteristics. With knowledge of the optimal solution to this problem, we tried to “help” the branching by adding some of the columns before the branching started. When we add ten (specific) columns to the partial solution before branch and bound, the algorithm finds the optimal solution in six seconds. Adding only nine of these ten columns results in a solution time over two hours. Another remarkable discovery considers the construction of the problem. The first 104 columns of the problems together form a solution to the set partitioning problem, where the rows sets R(j) are ordered sequentially, so J(1) = {1,2,…,6}, J(2) = {7,8,9,10}, J(3) = {11,12,…,16} and so on. The sizes of these columns differ. From these 106 columns, 62 are also contained in the optimal solution, 44 are not. If we remove these 44 columns, which is about 0.5% of the total number of columns, the problem is solved in four minutes. Examining these results further, we can solve the problem in 15 minutes by deleting 26 columns, or 0.3% and in 25 minutes by deleting 16 columns,

or 0.2% of the total number of columns. From these results we conclude that the difficulty of the problems are definitely not caused by the size of the problem, but by the problem structure. There is only a very small amount of columns that cause this extreme difficulty. Table 5.5 shows the characteristics of the dynamic constraint-based method on aa01 and aa04 when calculation is stopped after two hours. We will use these results for comparison in Sections 5.5.2 tot 5.5.4. For example, we can see whether the depth of the search tree decreases when we apply lower bounding techniques. Furthermore, we check the best solution found and the number of nodes checked in two hours.

Table 5.5: Characteristics of the basic branching method on aa01 and aa04

	aa01	**aa04**
Optimal solution	56137	26374
Cardinality optimal solution	102	66
Time (minutes)	120	120
Number of nodes	251,200,000	309,700,000
Minimum depth of tree (after 100 nodes)	48	16
Maximum depth of tree	104	48
Best solution found	INF	27324

5.5.2 Dual update heuristic during branch and bound

The first adjustment of the dynamic constraint-based branching method we consider is the use of the dual update heuristic during the branching procedure. While columns are sorted on increasing costs before branch and bound and we count the number of active columns per row on every node, the dual update heuristic can be applied relatively easy on a certain node. We simply check whether the reduced costs of the first active column to cover a certain row r is strictly positive. If this is the case, we can adjust the lower bound for all nodes that follow this node. However, we have to make sure that we undo this adjustment when the node is backtracked. We perform this extra procedure on every node. We will first check the influence of this adjustment on the performance of the dynamic constraint-based branching method on the test set, excluding aa01 and aa04. Recall from Section 5.1.2 that there are 33 problems for which we need branch and bound to determine the optimal solution. Table 5.6 shows the computing times of the dynamic constraint-based branching method with and without dual update heuristic on these problems. As we can see, the total time increases slightly when we perform the dual update heuristic on every node. This does not imply that the heuristic does not influence the branching; the total number of nodes decreases with 25%. In other words, the heuristic does achieve the required improvement in the branching procedure, meaning a decrease in the number of nodes that have to be checked, however the extra time effort needed is too large to offset the benefit of this improvement.

Table 5.6: Results of dynamic constraint-based branching with dual update heuristic

	Basic branching		Branching with dual update heuristic	
Name	**Time (s)**	**Nodes**	**Time (s)**	**Nodes**
nw32	0.00	8	0.00	9
nw40	0.00	3	0.00	3
nw39	0.00	6	0.00	5
nw20	0.00	11	0.00	9
nw23	0.00	12	0.00	10
nw34	0.00	4	0.00	4
Heart	0.03	2475	0.03	2475
nw42	0.00	11	0.00	11
Delta	0.02	614	0.00	614
nw25	0.00	8	0.00	8
nw38	0.00	6	0.00	6
nw35	0.00	4	0.00	4
nw36	0.00	27	0.00	24
Snowflake	2.34	42734	2.64	37456
Fives	0.02	188	0.02	188
Meteor	0.02	338	0.00	338
nw29	0.00	12	0.00	12
nw30	0.00	21	0.00	20
nw31	0.00	10	0.00	9
nw06	0.00	129	0.00	97
aa06	2.94	283242	2.69	203539
kl01	0.00	444	0.00	238
aa05	2.70	368061	3.03	252924
aa03	0.00	1960	0.02	1693
nw11	0.64	214073	0.81	188638
nw18	2.69	349927	2.25	215042
nw13	0.00	72	0.00	64
us04	0.00	28	0.00	25
kl02	0.47	121928	0.78	132375
nw03	0.00	45	0.00	37
nw04	0.28	9267	0.34	9869
nw17	0.00	141	0.03	160
us01	0.00	342	0.00	267
Total	**12.14**	**1396151**	**12.64**	**1046173**

Table 5.7 shows the characteristics of the branch and bound search for aa01 and aa04 when we perform the dual update heuristic on every node and stop calculation after 2 hours. This extra procedure does not seem to influence the branching results much. For both problems less nodes are checked and no better solution is found. For aa04, a slight improvement seems to be realized, since the depth of the branching tree does decrease with this adjustment. However, neither of the two problems is solved within two hours and for aa01 no solution is found at all.

Table 5.7: Characteristics of the branching method with dual update heuristic

	aa01	**aa04**
Optimal solution	56137	26374
Cardinality optimal solution	102	66
Time (minutes)	120	120
Number of nodes	241,600,000	275,300,000
Minimum depth of tree (after 100 nodes)	48	9
Maximum depth of tree	104	42
Best solution found	INF	27324

5.5.3 Dual 3OPT heuristic during branch and bound

In this section we examine the application of the dual 3OPT heuristic during branch and bound. Since applying the 3OPT heuristic requires more computational effort than the dual update heuristic, we can apply the heuristic on every node.

Table 5.8: Characteristics of the branching method with dual 3OPT heuristic

		aa01	aa04		aa01	aa04		aa01	aa04
Optimal solution		56137	26374		56137	26374		56137	26374
Cardinality optimal solution		102	66		102	66		102	66
Time (minutes)		120	120		120	120		120	120
Number of nodes		385,900,000	6,900,000		367,800,000	900,000		8,200	527,300
Minum depth of tree (after 100 nodes)		59	17		55	21		53	12
Maximum depth of tree		103	56		106	58		90	51
Best solution found		INF	27324		INF	27324		INF	27324
Number of times 3OPT when cardinality =	**1**	1	1	**2**	1	1	**5**	1	1
Number of times 3OPT when cardinality =	**2**	1	1	**4**	1	1	**10**	1	1
Number of times 3OPT when cardinality =	**3**	1	1	**6**	1	1	**15**	1	64
Number of times 3OPT when cardinality =	**4**	1	1	**8**	1	1	**20**	1	13757
Number of times 3OPT when cardinality =	**5**	1	1	**10**	1	1	**25**	1	64057
Number of times 3OPT when cardinality =	**6**	1	1	**12**	1	1	**30**	1	25573
Number of times 3OPT when cardinality =	**7**	1	1	**14**	1	1	**35**	1	244
Number of times 3OPT when cardinality =	**8**	1	1	**16**	1	1	**40**	5	0
Number of times 3OPT when cardinality =	**9**	1	1	**18**	1	9	**45**	1	0
Number of times 3OPT when cardinality =	**10**	1	1	**20**	1	29	**50**	1	0
Number of times 3OPT when cardinality =	**11**	1	1	**22**	1	23	**55**	6	0
Number of times 3OPT when cardinality =	**12**	1	1	**24**	1	213	**60**	305	0
Number of times 3OPT when cardinality =	**13**	1	1	**26**	1	3637	**65**	196	0
Number of times 3OPT when cardinality =	**14**	1	1	**28**	2	17814	**70**	111	0
Number of times 3OPT when cardinality =	**15**	1	1	**30**	1	34212	**75**	45	0
Number of times 3OPT when cardinality =	**16**	1	1	**32**	1	28550	**80**	410	0
Number of times 3OPT when cardinality =	**17**	1	3	**34**	2	15184	**85**	543	0
Number of times 3OPT when cardinality =	**18**	1	24	**36**	4	4697	**90**	11	0
Number of times 3OPT when cardinality =	**19**	1	139	**38**	1	827	**95**	0	0
Number of times 3OPT when cardinality =	**20**	1	333	**40**	1	233	**100**	0	0
Number of times 3OPT when cardinality =	**21**	1	885	**42**	1	118	**105**	0	0
Number of times 3OPT when cardinality =	**22**	1	1619	**44**	1	4	**110**	0	0
Number of times 3OPT when cardinality =	**23**	1	2627	**46**	1	0	**115**	0	0
Number of times 3OPT when cardinality =	**24**	1	4367	**48**	5	0	**120**	0	0
Number of times 3OPT when cardinality =	**25**	1	7128	**50**	5	0	**125**	0	0
Number of times 3OPT when cardinality =	**26**	1	10748	**52**	6	0	**130**	0	0
Number of times 3OPT when cardinality =	**27**	1	12819	**54**	1	0	**135**	0	0
Number of times 3OPT when cardinality =	**28**	1	13967	**56**	3	0	**140**	0	0
Number of times 3OPT when cardinality =	**29**	1	14458	**58**	10	0	**145**	0	0
Number of times 3OPT when cardinality =	**30**	1	13393	**60**	34	0	**150**	0	0

We use the following procedure:

- Apply the heuristic on thirty “layers” in the branching tree, in three different scenario’s:
    - When the number of variables in the solution equals 1,2,3,…,30
    - When the number of variables in the solution equals 2,4,6,…,60

  - When the number of variables in the solution equals 5,10,15,…,150
- Only apply the heuristic if the gap between the upper and lower bound is larger than one time the standard deviation of the costs of the columns
- Use the bound found in node n as a starting point in node (n+1)
- Stop after two hours

For these three scenario's, we report the minimum and maximum depth of the tree after the first 100 nodes, the total number of nodes searched, the best solution found and, for every one of the thirty "checkpoints", the number of times that we have performed the heuristic on this depth. Table 5.8 shows these results.
In the first two cases, a lot more nodes are checked in the same time for aa01. However, this does not lead to a decrease in the depth of the tree, nor to a better solution. For aa04, much less nodes are checked in all three scenario's, which is caused by the fact that the 3OPT heuristic has to be performed very often, varying from 684,944 times to 1,059,521 to 7,036,483 times. This implies that the bounds found are not much better than the bound before branch and bound. This conclusion is supported by the fact that the depth of the tree is hardly affected by the lower bounding techniques. Comparing the first and the third strategy, we see that performing the heuristic lower in the tree costs less time than performing it high in the tree. This is caused by the fact that the size of the induced subproblem is smaller deeper in the tree. For both problems no better solution is found within a runtime of two hours.

5.5.4 Lagrangian relaxation during branch and bound

In this section we examine the use of Lagrangian relaxation during branch and bound. We will consider the static convergent series method with $C^0 = 750$, $Q = 75$ and for two values of α, 0.99 and 0.99875. The procedure is the same as with the 3OPT dual heuristic:

- Apply the heuristic on thirty "layers" in the branching tree, in three different scenario's:
  - When the number of variables in the solution equals 1,2,3,…,30
  - When the number of variables in the solution equals 2,4,6,…,60
  - When the number of variables in the solution equals 5,10,15,…,150
- Only apply the heuristic if the gap between the upper and lower bound is larger than one time the standard deviation of the costs of the columns
- Use the bound found in node n as a starting point in node (n+1)
- Stop after two hours

In total, we tested six scenario's. Table 5.9 shows the results for $\alpha = 0.99$, Table 5.10 shows the results for $\alpha = 0.99875$.

Table 5.9: Characteristics of the branching method with conditional static convergent series (SCS) method, $\alpha = 0.99$

		aa01	aa04		aa01	aa04		aa01	aa04
Optimal solution		56137	26374		56137	26374		56137	26374
Cardinality optimal solution		102	66		102	66		102	66
Time (minutes)		120	43.83		120	40.78		120	120
Number of nodes		285,900,000	126,267,005		131,600,000	92,495,691		3,055,800	3,658,900
Minum depth of tree (after 100 nodes)		49	0		29	0		59	2
Maximum depth of tree		100	66		99	66		100	65
Best solution found		60293	26374		57717	26374		56768	26677
Proven optimal?		no	yes		no	yes		no	no
Number of times SCS when cardinality =	**1**	1	3	**2**	1	12	**5**	1	38
Number of times SCS when cardinality =	**2**	1	9	**4**	1	55	**10**	1	1534
Number of times SCS when cardinality =	**3**	1	15	**6**	1	108	**15**	1	5465
Number of times SCS when cardinality =	**4**	1	33	**8**	1	230	**20**	1	8201
Number of times SCS when cardinality =	**5**	1	52	**10**	1	264	**25**	1	2836
Number of times SCS when cardinality =	**6**	1	77	**12**	1	603	**30**	1	195
Number of times SCS when cardinality =	**7**	1	118	**14**	1	765	**35**	1	2
Number of times SCS when cardinality =	**8**	1	170	**16**	1	606	**40**	1	0
Number of times SCS when cardinality =	**9**	1	183	**18**	1	310	**45**	1	0
Number of times SCS when cardinality =	**10**	1	172	**20**	1	247	**50**	1	0
Number of times SCS when cardinality =	**11**	1	261	**22**	1	142	**55**	1	0
Number of times SCS when cardinality =	**12**	1	291	**24**	1	42	**60**	4	0
Number of times SCS when cardinality =	**13**	1	198	**26**	1	2	**65**	190	0
Number of times SCS when cardinality =	**14**	1	97	**28**	1	2	**70**	976	0
Number of times SCS when cardinality =	**15**	1	73	**30**	2	11	**75**	4771	0
Number of times SCS when cardinality =	**16**	1	93	**32**	22	2	**80**	18603	0
Number of times SCS when cardinality =	**17**	1	83	**34**	69	2	**85**	729	0
Number of times SCS when cardinality =	**18**	1	89	**36**	122	2	**90**	2	0
Number of times SCS when cardinality =	**19**	1	103	**38**	178	2	**95**	2	0
Number of times SCS when cardinality =	**20**	1	80	**40**	225	2	**100**	5	0
Number of times SCS when cardinality =	**21**	1	43	**42**	178	2	**105**	0	0
Number of times SCS when cardinality =	**22**	1	39	**44**	267	2	**110**	0	0
Number of times SCS when cardinality =	**23**	1	25	**46**	383	2	**115**	0	0
Number of times SCS when cardinality =	**24**	1	31	**48**	513	2	**120**	0	0
Number of times SCS when cardinality =	**25**	1	35	**50**	954	2	**125**	0	0
Number of times SCS when cardinality =	**26**	1	25	**52**	1554	2	**130**	0	0
Number of times SCS when cardinality =	**27**	1	13	**54**	2149	2	**135**	0	0
Number of times SCS when cardinality =	**28**	1	2	**56**	2472	2	**140**	0	0
Number of times SCS when cardinality =	**29**	1	3	**58**	2977	2	**145**	0	0
Number of times SCS when cardinality =	**30**	1	0	**60**	3151	2	**150**	0	0

We first consider the results for aa01. In all six scenario's, a solution is found for aa01, ranging from 56,138, or 0.002% from the optimum, to 60,293, or 7.4% from the optimum. In none of the six scenario's the optimum is reached. The depth of the branch and bound tree is indeed influenced by the use of Lagrangian relaxation during branch and bound; the minimum depth of the tree after the first 100 nodes ranges from 22 to 59 nodes. Obviously, the more often we perform the subgradient search procedure, the longer time we need and the less time we have left for the branching. This explains why, for $\alpha = 0.99$, we can search 285,900,000 nodes in the first scenario, where subgradient search is only performed 30 times, against 3,055,800 nodes in the third scenario, where subgradient search is performed 25,293

times. The performance of the branch and bound procedure is better when we use $\alpha = 0.99875$ than when we use $\alpha = 0.99$. This can be concluded since more nodes are searched, the depth of the tree is smaller and the solutions found are better.

We now consider the results for problem aa04. In three of the six scenario's, this problem is solved to optimality in time ranging from 41 to 82 minutes. The best setting, where aa04 is solved in 41 minutes, is to perform subgradient search on every even node with $\alpha = 0.99$. Performing subgradient search every five nodes does not work out very good for aa04, since much time is lost in calculating lower bounds. In all six scenario's, the best solution found is less than 2% away from the optimum.

Table 5.10: Characteristics of the branching method with conditional static convergent series method, $\alpha = 0.99875$

		aa01	aa04		aa01	aa04		aa01	aa04
Optimal solution		56137	26374		56137	26374		56137	26374
Cardinality optimal solution		102	66		102	66		102	66
Time (minutes)		120	81.98		120	120		120	120
Number of nodes		340,900,000	170,973,332		281,600,000	4,900,000		111,700,000	3,600,000
Minum depth of tree (after 100 nodes)		23	0		31	10		22	6
Maximum depth of tree		101	66		99	66		100	68
Best solution found		58152	26374		56770	26771		56138	26908
Proven optimal?		no	yes		no	no		no	no
Number of times SCS when cardinality =	**1**	1	3	**2**	1	1	**5**	1	1
Number of times SCS when cardinality =	**2**	1	6	**4**	1	1	**10**	1	180
Number of times SCS when cardinality =	**3**	1	14	**6**	1	1	**15**	1	3082
Number of times SCS when cardinality =	**4**	1	29	**8**	1	1	**20**	1	392
Number of times SCS when cardinality =	**5**	1	35	**10**	1	1	**25**	38	115
Number of times SCS when cardinality =	**6**	1	55	**12**	1	8	**30**	319	0
Number of times SCS when cardinality =	**7**	1	84	**14**	1	136	**35**	527	0
Number of times SCS when cardinality =	**8**	1	105	**16**	1	465	**40**	771	0
Number of times SCS when cardinality =	**9**	1	111	**18**	1	565	**45**	446	0
Number of times SCS when cardinality =	**10**	1	113	**20**	1	605	**50**	1	0
Number of times SCS when cardinality =	**11**	1	112	**22**	1	267	**55**	1	0
Number of times SCS when cardinality =	**12**	1	161	**24**	1	345	**60**	72	0
Number of times SCS when cardinality =	**13**	1	113	**26**	1	188	**65**	71	0
Number of times SCS when cardinality =	**14**	1	75	**28**	1	26	**70**	133	0
Number of times SCS when cardinality =	**15**	1	37	**30**	1	0	**75**	196	0
Number of times SCS when cardinality =	**16**	1	28	**32**	7	0	**80**	39	0
Number of times SCS when cardinality =	**17**	1	25	**34**	22	0	**85**	262	0
Number of times SCS when cardinality =	**18**	1	7	**36**	70	0	**90**	297	0
Number of times SCS when cardinality =	**19**	1	6	**38**	140	0	**95**	3	0
Number of times SCS when cardinality =	**20**	1	9	**40**	90	0	**100**	5	0
Number of times SCS when cardinality =	**21**	1	2	**42**	98	0	**105**	0	0
Number of times SCS when cardinality =	**22**	1	3	**44**	110	0	**110**	0	0
Number of times SCS when cardinality =	**23**	1	1	**46**	99	0	**115**	0	0
Number of times SCS when cardinality =	**24**	2	1	**48**	100	0	**120**	0	0
Number of times SCS when cardinality =	**25**	7	1	**50**	31	0	**125**	0	0
Number of times SCS when cardinality =	**26**	14	1	**52**	36	0	**130**	0	0
Number of times SCS when cardinality =	**27**	12	1	**54**	51	0	**135**	0	0
Number of times SCS when cardinality =	**28**	13	1	**56**	39	0	**140**	0	0
Number of times SCS when cardinality =	**29**	14	1	**58**	33	0	**145**	0	0
Number of times SCS when cardinality =	**30**	15	1	**60**	16	0	**150**	0	0

We can conclude that indeed some improvement is realized for our two problem cases by using subgradient search during branch and bound, however the performance is still not very good. Both aa01 and aa04 are solved by CPLEX within two minutes. In general, calculating lower bounds during branch and bound can help somewhat in solving the difficult instances, but the time needed is quite large. Table 5.11 shows the results of Lagrangian relaxation during branch and bound, with α = 0.99 and lower bound calculation on nodes 2,4,6,…,60 for the problems in our test set that are not solved before branch and bound. The performance of the branch and bound procedure is hardly affected: the total time increases with 8.5% from 12 to 13 seconds and the total number of nodes decreases slightly with 4.7%.

Table 5.11: Results of dynamic constraint-based branching with conditional static convergent series method on nodes 2,4,6,…,60 and with α = 0.99

	Basic branching		Branching with subgradient search	
Name	**Time (s)**	**Nodes**	**Time (s)**	**Nodes**
nw32	0.00	8	0.00	8
nw40	0.00	3	0.00	3
nw39	0.00	6	0.00	6
nw20	0.00	11	0.00	11
nw23	0.00	12	0.00	12
nw34	0.00	4	0.00	4
Heart	0.03	2475	0.63	0
nw42	0.00	11	0.00	11
Delta	0.02	614	0.33	0
nw25	0.00	8	0.00	8
nw38	0.00	6	0.00	6
nw35	0.00	4	0.00	4
nw36	0.00	27	0.00	27
Snowflake	2.34	42734	2.59	42734
Fives	0.02	188	0.08	0
Meteor	0.02	338	0.05	0
nw29	0.00	12	0.00	12
nw30	0.00	21	0.00	21
nw31	0.00	10	0.00	10
nw06	0.00	129	0.00	129
aa06	2.94	283242	1.91	186115
kl01	0.00	444	0.00	444
aa05	2.70	368061	3.14	415415
aa03	0.00	1960	0.02	1960
nw11	0.64	214073	0.72	214073
nw18	2.69	349927	2.88	349927
nw13	0.00	72	0.00	119
us04	0.00	28	0.00	28
kl02	0.47	121928	0.45	109217
nw03	0.00	45	0.00	45
nw04	0.28	9267	0.36	9267
nw17	0.00	141	0.03	141
us01	0.00	342	0.00	461
Total	**12.14**	**1,396,151**	**13.17**	**1,330,218**

5.6 Concluding remarks

In this chapter we discussed several branching strategies to solve set partitioning problems. We considered the classical variable-based branching, which results in a binary search tree. Furthermore, we examined several constraint-based branching strategies, where we take a row-wise approach. The row orderings in this strategy can be static or dynamic and we considered several possibilities. Comparing the computational results on our test set, the dynamic constraint based branching method performs best. This approach considers the columns sorted on increasing reduced costs and, in every iteration, chooses the row which is covered by the least number of active columns to branch next. This strategy performs well on all instances in our test set, except two, which are not solved to optimality in a reasonable amount of time in all methods considered. The strength of the branch and bound method used in LaRSS, the dynamic constraint based branching method, is that the search itself can be performed very quickly, since preprocessing and bound calculations are all done before branching. However, for problems with difficult cost structures, this strength turns out to be a weakness, since the quality of the bounds is not good enough to find the optimum fast.

To improve the performance of the dynamic constraint-based branching method, especially on the two instances not solved, we considered several lower bounding techniques that can be used during branch and bound. Considering the two dual heuristics, the quality of the lower bounds found appears to be not good enough to offset the time needed to calculate them. The use of subgradient search techniques during branch and bound do seem to have a positive impact on the branching procedure for the two problematic instances; the depth of the tree decreases for both instances and the solutions found are better. Moreover, in some scenario's considered, one of these instances is solved in reasonable time, ranging from 41 to 82 minutes.

When we perform subgradient search on the nodes 2,4,6,…,60 with $\alpha = 0.99$, one instance, aa04, is solved to optimality within 41 minutes. The other instance, aa01, is not solved to optimality in two hours, but the best solution found is less than 3% away from the optimal solution. For the other instances, the extra lower bound calculation leads to a slight increase in average computing time of 5%. In LaRSS, the dynamic constraint-based method is used without extra lower bound calculations.

Chapter 6

Miscellaneous research results

This chapter deals with miscellaneous research results. It is divided into two parts. The first part briefly considers cuts for set partitioning problems without the use of linear programming information. The second part considers research on a possible decomposition approach to solve set partitioning problems.

6.1 Cuts

This section briefly examines the possibilities of adding cuts for our set partitioning algorithms. We do not intend to give an extensive discussion of cuts and polyhedral theory. For more information on the theory we refer to Schrijver (1999).

A linear constraint that does not exclude any integer feasible points to a certain integer programming problem, is called a cut or cutting plane (Papadimitriou and Steiglitz (1982). In literature, much information can be found on cutting planes for set partitioning problems. Mostly, these cuts are used in algorithms that are based on linear programming relaxations, while linear programming information is of great importance for determining the value of a certain cut. Essentially, the purpose of adding a cut to a set partitioning problem is to cut of a part of the solution set of the LP relaxation of the problem without cutting of any feasible integer solution, such that the lower bound increases. When we have a linear programming solver at our disposal and we know the basis and solution of the current linear programming relaxation problem, it is easily checked if the cut we want to add will help in increasing this solution. Moreover, linear programming software can help to lift the inequalities that are found. However, when we have a Lagrangian relaxation based algorithm, there is no way to check whether adding a cut helps to solve the problem faster, except by actually solving it.

For more information about cuts for set partitioning problems we refer to Balas

and Padberg (1976), Hoffman and Padberg (1993) and Borndörfer (1999). For illustrative purpose, we will only consider one type of cut, that is discussed in each one of these references: the clique inequalities.

6.1.1 Clique inequalities

The most widely used cuts are the clique inequalities, since they define facets, are easy to implement, numerically stable and sparse (Borndörfer, 1999). Another advantage for our use is that the resulting inequalities are set packing constraints, which we can easily handle in our set partitioning solver by using slack variables. Formally, the clique inequalities are defined as follows (Balas and Padberg, 1977). Let G_A = (V,E) be the intersection graph of matrix A, where every vertex v in V represents a column of the matrix A and two vertices v_1 and v_2 are connected by an edge e = (v_1,v_2) if the columns corresponding to these vertices are nonorthogonal, meaning that they have no nonzero elements in common. Now the inequality

$$\sum_{j \in K} x_j \leq 1 \qquad [6.1]$$

where K is the node set of a clique in G_A, is a cut on the set partitioning problem with constraint matrix G. Amongst others, Hoffman and Padberg (1993) and Borndörfer (1999) use these constraints to improve the performance of their algorithms. They use linear programming software to determine the value of the cuts and to lift them. We will give two examples of greedy heuristics to illustrate the difficulty in using these cuts in our Lagrangian relaxation setting.

6.1.2 Two heuristics for determining clique cuts

We examine two greedy heuristics for generating clique cuts in LaRSS. Figure 6.1 shows the heuristic 1. In this heuristic we use a certain column set H, to be determined before the start of the heuristic. We then greedily add columns to a clique. Below we give an example with five rows and five columns, where H = J.

Example

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Cut: $x_1 + x_2 + x_4 + x_5 \leq 1$

Figure 6.1: Heuristic 1 for determining clique cuts

Consider a set of columns $H \subseteq J$, to be determined before the start of the heuristic.
For every column h in H
previous_row = -1
next_col = h
Empty stack
While next_col > 0
Add next_col to the stack
rowset = $\{r \in R(\text{next\_col}) \mid r > \text{previous\_row}\}$
If rowset $\neq \emptyset$
next_row = $\min\{r \mid r \in \text{rowset}\}$
previous_row = next_row
colset = $\left\{ j \in J(\text{next\_row}) \mid j \notin \text{stack}, \sum_r a_{rj} \cdot a_{rk} > 0 \forall k \in \text{stack} \right\}$
If colset $\neq \emptyset$
next_col = $\min\{j \mid j \in \text{colset}\}$
Else
next_col = -1
EndIf
EndIf
EndWhile
Write a clique cut with “1” for all columns on the stack
EndFor

Heuristic 1, given in Figure 6.2, is a revised version of the greedy heuristic discussed by Borndörfer (1999), page 134. Borndörfer uses this heuristic in a larger set of cut generating heuristics. For the original heuristic we need the values of the primal solution of the LP relaxation of the set partitioning problem. Since these values are not available, we use pseudo-primal solutions, that are constructed from the SCS subgradient search method in the following way, which is based on the volume algorithm of Barahona and Anbil (2000). Suppose x^k is the solution vector in the k^{th} iteration of the SCS method. Now we define the pseudo primal solution to be $\tilde{x}^k$, which we update according to the following scheme:

$\tilde{x}^0 = 0$

$\tilde{x}^{k+1} = 0.95 \cdot \tilde{x}^k + 0.05 \cdot x^{k+1}$

After we have performed the SCS method to calculate these value, we can use the following heuristic to determine the cuts.

Let F be the set of columns {f} for which $0 < \tilde{x}_f < 1$ and let F be sorted in such a way

that $F = \{f_1,...,f_{|F|}\}$ with $\tilde{x}_{f_1} \geq \tilde{x}_{f_2} \geq ... \geq \tilde{x}_{f_{|F|}}$.

Figure 6.2: Heuristic 2 for determining clique cuts

For every column f_k in F
  stack = $\{f_k\}$
  For every column g in the set $\{f_{k+1},\ldots,f_{|F|}\}$
    If $\sum_r a_{rg} \cdot a_{rf} > 0 \forall f \in$ stack
      Add g to stack
    EndIf
  EndFor
  $\Gamma(\text{stack}) = \left\{ j \in J \setminus \text{stack} \mid \sum_r a_{rj} \cdot a_{sj} > 0 \forall s \in \text{stack} \right\}$
  $s = \arg\max_{r \in R} \left(\left| J(r) \cap \Gamma(\text{stack}) \right| \right)$
  $\text{stack} = \text{stack} \cup \left(J(s) \cap \Gamma(\text{stack}) \right)$
  Write a clique cut with “1” for all columns on the stack
EndFor

6.1.3 Computational results

We will consider computational experiments for both heuristics. In the first heuristic we have to determine the column set H. Taking all columns, or H = J, would result in far too many cuts to handle, so we need some criterion to determine which columns to use. To have more information available to take this decision, we use the following setting to test these heuristics. Similar to the setup of LaRSS, see Sections 3.5.2 and 7.1, we perform two subgradient search procedures. The first is a quick and simple application of the SCS method, the second is the more detailed DCS method to find the lower bound used in the branch and bound routine. In the first lower bound method, we can acquire information like the pseudo-primal solution as discussed in the previous section, but also information about the reduced costs of the columns. We can use this extra information to determine the cuts before the DCS method.

Table 6.1 shows some computational results of the first heuristic on five of the problems in our test set. We have taken the American Airlines instances to illustrate the results, however aa02 is left out, since the DCS already finds the optimal solution for this problem. We have chosen to use three different strategies:

- H is the set of all columns with fractional pseudo-primal solution value; add all cuts.
- H is the set of all columns with fractional pseudo-primal solution value; add cuts only if the number of nonzero’s is larger than the 75% of the average row density.

- H is the set of all columns that have 0 reduced costs after the first subgradient search.

Table 6.1: Results of heuristic 1 on five problems from our test set

H: set of columns with fractional pseudo-primal solution value				
	LB without cuts	Number of cuts	Average size cut	LB with cuts
aa01	55519.00	770	11.956	55481.93
aa03	49607.10	750	12.244	49610.15
aa04	25870.36	791	16.044	25852.19
aa05	53721.42	701	11.097	53712.15
aa06	26973.26	954	12.547	26959.57
H: set of columns with fractional pseudo-primal solution value Add cut only if size is larger than 75% of the average number of nonzero's per row				
	LB without cuts	Number of cuts	Average size cut	LB with cuts
aa01	55519.00	14	48.5	55521.20
aa03	49607.10	24	51.917	49608.74
aa04	25870.36	19	87.421	25871.30
aa05	53721.42	22	46.818	53693.82
aa06	26973.26	38	58.842	26952.71
H: set of columns with $cr_j = 0$				
	LB without cuts	Number of cuts	Average size cut	LB with cuts
aa01	55519.00	183	12.699	55511.50
aa03	49607.10	155	12.374	49612.70
aa04	25870.36	120	15.65	25870.08
aa05	53721.42	161	11.776	53709.18
aa06	26973.26	138	11.232	26973.03

Table 6.2: Results of heuristic 2 on five problems from our test set

Add all cuts				
	LB without cuts	Number of cuts	Average size cut	LB with cuts
aa01	55519.00	770	130.977	55527.07
aa03	49607.10	750	140.719	49601.97
aa04	25870.36	791	179.023	25882.61
aa05	53721.42	701	148.636	53701.10
aa06	26973.26	954	129.361	26971.82
Add cut only if size is larger than 75% of the average number of nonzero's per row				
	LB without cuts	Number of cuts	Average size cut	LB with cuts
aa01	55519.00	672	145.286	55525.70
aa03	49607.10	670	152.796	49612.03
aa04	25870.36	611	214.083	25871.77
aa05	53721.42	643	158.894	53719.73
aa06	26973.26	790	148.030	26944.91

For all of these three strategies, the lower bound decreases for at least one of the

problems, but also increases for at least one of the instances. The differences in bounds are very small for all instances. Before we analyze these results, we first consider the results for the second heuristic on these five instances.

Table 6.2 shows some computational results of the second heuristic on five of the problems in our test set. Again, we see a very divers set of results, where sometimes the lower bound increases after we add the cuts, but also sometimes the bound decreases. Note that we aim to illustrate the results of our research by these examples; the results on the whole test set and for other strategies give the same impression. Performing these operations have two negative effects on the computing times. First, the computing time of the heuristics cause an increase in the total time needed. Second, the computing times of the subgradient search and branch and bound algorithms increase due to the larger number of rows. We would expect this increase in computing times to be compensated by an increase in the lower bound and thus a decrease in the time needed for the branch and bound procedure, which is not the case. In the next section we summarize and analyze these results.

6.1.4 Concluding remarks

In this section, we have introduced the concept of cuts and illustrated the difficulties in using this concept in a Lagrangian relaxation-context. The important difficulty of this approach is the lack of information about the LP solution. When we have a solution and a basis for the LP relaxation, we can use this information to create and evaluate cuts. After we add a cut, we can immediately see if this influences the LP solution. However, when we have a Lagrangian relaxation (LR) approach, we have no information to start with and thus cannot decide which variables can be of use. Moreover, when we have added one or more cuts, the only way to see if the solution to the LR problem increases is to solve it once again. However, the subgradient search methods we use to solve the LR problem are heuristic and thus a decrease in the lower bound does not necessarily imply that the cut does not cut of any fractional solution. With an LP solver available, cuts can be found and evaluated very easily.

We have illustrated these difficulties by two heuristics to create clique cuts in our settings. For different strategies we have analyzed the influence of the cuts found on the lower bounds. These results are not very stable: for all five strategies we tested, neither was useful for all five problems. Moreover, the differences in the bounds are very small.

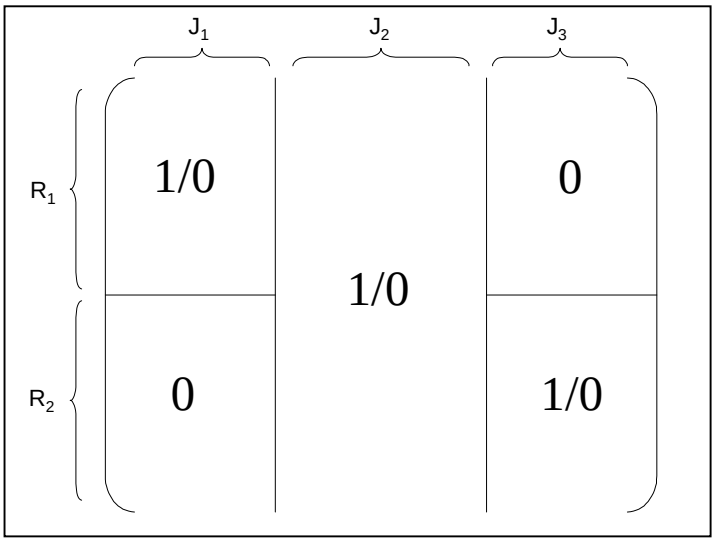
6.2 Decomposition approach

This section examines two possibilities to decompose a set partitioning problem into smaller, easier to solve sub-problems.

6.2.1 Basic concept

Suppose a set partitioning problem is given with row set R, column set J and constraint matrix A. We now want to interchange rows and columns in such a way that the structure depicted in Figure 6.3 is obtained.

Figure 6.3: Concept of the decomposed constraint matrix



In other words, we decompose the row set in two sets, R_1 and R_2 and the column set in three sets J_1, J_2 and J_3, with two restrictions:

1. $\forall r \in R_2, \forall j \in J_1 : a_{rj} = 0$ [6.2]
2. $\forall r \in R_1, \forall j \in J_3 : a_{rj} = 0$ [6.3]

Note that for every partition of the rows in two sets R_1 and R_2, the decomposition of the columns in the sets J_1, J_2 and J_3 follows naturally. If the set partitioning tableau is decomposed in this way, we could use the special structure to solve the set partitioning problem faster. This could be done, for example, by first solving the set partitioning problem formed by R_1 and J_1 and the set partitioning problem formed by R_2 and J_3. Then, we would not only have a solution and thus an upper bound to the problem, but also we could use the information about J_1 and J_3 in the branching of the columns in J_2. This is particularly interesting if the number of columns in J_2 is relatively small.

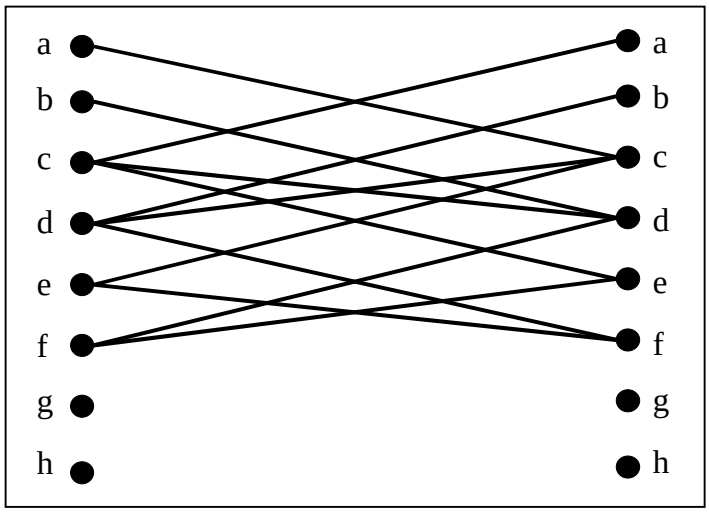
6.2.2 Problem formulation

We will use a graph theoretic approach to formulate the problem of finding the proposed decomposition. We construct a bipartite graph $G = ((V_1 \cup V_2), E)$, where V_1 and V_2 both represent the columns of the tableau, so $|V_1| = |V_2| = |J|$, and $e = (v, w) \in E$ for $v \in V_1, w \in V_2$ if the columns corresponding to vertices v and w have no nonzero elements in common in the set partitioning tableau.
For example, consider the following set partitioning constraint matrix:

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

The proposed graph is given in Figure 6.4, where a,…,h corresponds to the columns of the constraint matrix.

Figure 6.4: Example of the proposed graph construction



A biclique in the graph G is formed by a subgraph G', $G' = ((V'_1 \cup V'_2), E')$, with $V'_1 \subseteq V_1$, $V'_2 \subseteq V_2$ and $E' \subseteq E$, which forms a complete bipartite graph. Note that such a biclique corresponds to two sets of columns of the set partitioning tableau that have no nonzero elements in common. Given a biqlique G', we can define a corresponding decomposition of the set partitioning tableau in the following way:

1. $J_1 = V'_1$
2. $J_3 = V'_2$

3. $J_2 = V \setminus \{V'_1 \cup V'_2\}$
4. $R_1 = \{r \in R \mid \forall j \in J_3, a_{rj} = 0\}$
5. $R_2 = \{r \in R \mid \forall j \in J_1, a_{rj} = 0\}$

In the example, a biclique is formed by V'_1 = {d,e} and V'_2 = {c,f}. With some column interchanging, we find the decomposed constraint matrix A':

$$A' = \left[\begin{array}{cc:cccc:cc} d & e & a & b & g & h & c & f \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{array}\right]$$

Obviously, the proposed bipartite graph G could have multiple bicliques. To make optimal use of the decomposition, however, we want to find a decomposition such that the number of columns in J_2 is as small as possible. Moreover, we would like to have a balanced decomposition, such that the number of columns in J_1 and J_3 is approximately equal. To accomplish the first goal, we would like to find the largest biclique in G, such that $|V'_1| + |V'_2|$ is maximized. To achieve the second goal, one could propose to find the biclique which maximizes $|V_1| x |V_2|$. However, this problem, known as the maximum edge biclique problem, can be shown to be NP-complete (Peeters, 2000), while the former problem, the maximum node biclique problem, is solvable in polynomial time (Hochbaum, 1998). This maximum node biclique problem is defined as follows, where $V = V_1 \cup V_2$ and the decision variable $x_j, j \in V$ indicates whether node j is in the biclique.

$$\text{Max} \quad \sum_{j \in V} x_j \qquad [6.4]$$

Subject to:

$$x_i + x_j \leq 1 \qquad (i,j) \notin E, \quad i \in V_1, j \in V_2 \qquad [6.5]$$

$$x_j \in \{0,1\} \qquad j \in V \qquad [6.6]$$

While the constraint matrix of this formulation is totally unimodular, this problem can be solved in polynomial time. We must however, take some precaution, since the optimal solution for this problem is obviously given by a degenerate biclique, where we take all vertices from V_1 and no vertices from V_2 (or the other way around). We can take care of this problem, for example by demanding that at least m vertices of V_1 and at least m vertices of V_2 must be taken into the biclique, or:

$$\sum_{j \in V_1} x_j \geq m \qquad [6.7]$$

$$\sum_{j \in V_2} x_j \geq m \qquad [6.8]$$

Here, m > 0 and if m increases, the size of the biclique decreases, so m = 1 corresponds to the largest biclique in terms of number of nodes. The problem of

finding the biclique B in G that maximizes $|V_1|+|V_2|$ is called the node biclique problem on bipartite graphs. This problem is equivalent to the maximum independent set on bipartite graphs, which is known to be solvable by a minimum cut algorithm (Hochbaum, 1998).

6.2.3 Computational experiments

To examine the possibilities of the decomposition approach, we used CPLEX to solve the mathematical programming problem of [6.4] – [6.8] in order to determine the maximal node biclique, under the restrictions [6.7] and [6.8], for the set partitioning instances in our test set. Since these problems become very large when the number of columns of the problem increases, we first examine the 15 smallest problems in the test set. Table 6.1 shows the results for these problems, where the calculation is stopped when the time exceeds 10 minutes. In this case, the best solution found is shown.

Table 6.3: Decompositions for different values of m

		m = 1			m = 5			m = 10			m = 15			m = 20		
	# Cols	$\|J_1\|$	$\|J_3\|$	Time (s)	$\|J_1\|$	$\|J_3\|$	Time (s)	$\|J_1\|$	$\|J_3\|$	Time (s)	$\|J_1\|$	$\|J_3\|$	Time (s)	$\|J_1\|$	$\|J_3\|$	Time (s)
nw41	197	186	1	256.30	146	5	185.99	124	13	> 600	94	15	> 600	71	20	> 600
nw32	294	293	1	3.13	204	5	293.95	93	10	> 600	61	16	> 600	41	20	> 600
nw40	404	370	1	> 600	200	5	403.44	93	12	> 600	70	44	> 600	70	44	> 600
nw08	434	421	3	> 600	414	6	434.00	365	10	> 600	160	15	> 600	88	21	> 600
nw15	467	466	1	10.84	372	5	467.00	121	10	> 600	60	15	> 600	0	0	> 600
nw21	577	558	1	> 600	238	5	576.47	276	12	> 600	92	23	> 600	92	23	> 600
nw22	619	376	3	> 600	179	7	> 600	269	17	> 600	57	16	> 600	38	22	> 600
nw12	626	602	24	25.11	602	24	> 600	602	24	25.3	602	24	25.3	602	24	25.25
nw39	677	637	1	> 600	381	5	> 600	112	10	> 600	93	16	> 600	35	24	> 600
nw20	685	677	1	> 600	398	5	> 600	136	10	> 600	53	16	> 600	89	45	> 600
nw23	711	710	1	47.61	491	5	> 600	685	10	> 600	447	15	> 600	222	20	> 600
nw37	770	756	1	> 600	424	5	> 600	241	10	> 600	153	18	> 600	112	26	> 600
nw26	771	714	2	> 600	699	41	> 600	699	41	> 600	699	41	> 600	699	41	> 600
nw10	853	840	1	> 600	778	5	> 600	93	36	> 600	93	36	> 600	93	36	> 600
nw34	899	354	2	> 600	102	6	> 600	206	10	> 600	118	15	> 600	33	20	> 600

To make use of the decomposition to reduce the solution time of the problem, we want to have a decomposition where the number of columns in J_2, $|J_2| = |J| - |J_1| - |J_3|$, is small, while the remaining columns are divided equally over J_1 and J_3. Otherwise stated, we want to have a balanced decomposition. For m = 1 we see that the number of columns in J_2 is small for all 15 instances, 9% on average, but also that the number of columns J_1 is relatively large, 90% on average. The decompositions become more balanced when m increases, however also the number of columns in J_2 increases. When m = 20, the number of columns in J_2 is 70% on

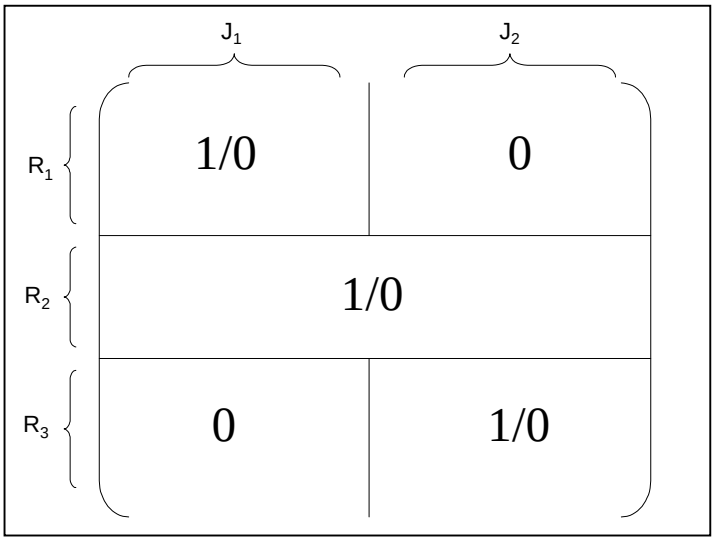
average. The computing times of these problems is quite long, which is expected since the number of variables and constraints of the problems is very large. If the decomposition idea is to be implemented, one would have to develop a heuristic to determine appropriate decompositions.

For the problems recorded in Table 6.3, decompositions would not be very useful, since these problems are solved very fast by a regular set partitioning solver. To further investigate the use of decompositions, we thus examine another subset of our test set of problems. In this case we examine the six “American Airlines” (aa) instances. The solution times of these problems with LaRSS are relatively long, and two of them are not solved within 10 minutes. Using CPLEX to determine the optimal decomposition we find that none of these problems can be decomposed in the way depicted in Figure 6.3 when we demand that the number of columns in J_1 and J_2 are strictly positive. This is concluded within 10 minutes for all six instances. Combining these findings and the results in Table 6.3, we conclude that this decomposition idea is not useful to reduce the solution time of set partitioning problems.

6.2.4 An alternative decomposition approach

So far, this section considered a column-wise decomposition of the set partitioning tableau, as depicted in Figure 6.3. We will now examine another decomposition approach, that is closely related to the decomposition discussed above. Suppose again a set partitioning problem is given with row set R, column set J and constraint matrix A. We now want to interchange rows and columns in such a way that the structure depicted in Figure 6.5 is obtained.

Figure 6.5: Concept of the alternative decomposition approach



In other words, we decompose the column set in two sets, J_1 and J_2 and, accordingly,

the row set in three sets R_1, R_2 and R_3, with two restrictions:

1. $\forall r \in R_3, \forall j \in J_1 : a_{rj} = 0$ [6.9]
2. $\forall r \in R_1, \forall j \in J_2 : a_{rj} = 0$ [6.10]

We can now apply the theory of Section 6.2.2 on this "tilted" decomposition to get the following problem formulation. Again, we construct a bipartite graph $G = ((V_1 \cup V_2), E)$. This time, V_1 and V_2 both represent the rows of the tableau, so $|V_1| = |V_2| = |R|$, and $e = (v, w) \in E$ for $v \in V_1, w \in V_2$ if the rows corresponding to vertices v and w have no nonzero elements in common in the set partitioning tableau. With these new definitions for E and V, the formulas [6.4] to [6.6] give the problem formulation of finding the maximum biclique in the graph G. The maximum biclique corresponds to a row-wise decomposition in which the number of rows in R_1 and R_3 is maximal. Again, we add constraints [6.7] and [6.8] to enforce that at least m rows are taken into R_1 and R_3.

We illustrate the value of the alternative decomposition approach by showing the results of CPLEX of solving this mathematical programming problem for the six American Airline instances in our test set, for different values of m. These results are given in Table 6.2.

Table 6.4: Row-wise decompositions for different values of m

		m = 20			m = 40			m = 60			m = 80			m = 100		
	# Rows	$\|R_1\|$	$\|R_3\|$	Time (s)	$\|R_1\|$	$\|R_3\|$	Time (s)	$\|R_1\|$	$\|R_3\|$	Time (s)	$\|R_1\|$	$\|R_3\|$	Time (s)	$\|R_1\|$	$\|R_3\|$	Time (s)
aa01	823	377	20	> 600	40	319	> 600	60	141	> 600	80	146	> 600	100	122	> 600
aa02	531	20	444	> 600	339	40	> 600	60	196	> 600	80	133	> 600	0	0	> 600
aa03	825	20	672	> 600	40	372	> 600	255	60	> 600	161	80	> 600	101	106	> 600
aa04	426	241	20	> 600	60	44	> 600	0	0	> 600	0	0	> 600	0	0	> 600
aa05	801	20	583	> 600	40	479	> 600	60	261	> 600	80	172	> 600	100	105	> 600
aa06	646	493	20	> 600	312	40	> 600	177	61	> 600	85	91	> 600	0	0	> 600

We see that the size of the biclique decreases when the value of m increases. When m is small, the bicliques found are not balanced, where balanced means that the number of rows in R_1 and R_3 are of the same order. For m = 100, a decomposition is only found for three of the instances. These decompositions are well balanced, however, they are too small to be of value, since the number of rows in the set R_2 is too large. In all three cases, about 75% of the rows is in R_2. In all 30 cases examined, the decomposition problem was not solved to optimality within 10 minutes. Again, we conclude that this decomposition approach is not useful to be applied in LaRSS.

Chapter 7

LaRSS

This chapter describes LaRSS, our solver for pure set partitioning problems that uses a combination of the techniques discussed in this thesis. First, Section 7.1 considers the construction of LaRSS: which techniques are used and in which sequence. Then, Section 7.2 examines the performance of LaRSS on our test set and compares these results those of CPLEX. Section 7.3 considers some technical aspects considering the methods used in LaRSS. Finally, Section 7.4 concludes this Chapter.

7.1 Construction of LaRSS

All techniques used in LaRSS are discussed in Chapters 2 to 5. Obviously, there are many interactions between the different methods. For example, performing preprocessing techniques before determining the lower bound can reduce the calculating time considerably. On the other hand, knowledge of lower- and upper bounds can reduce the calculating time of the row combination techniques, since combined columns only have to be added to the tableau if their costs do not exceed the upper bound minus the lower bound. The sequence in which methods are applied thus must be determined very carefully. The sequence used in LaRSS is a result of empirical research, taking into account these dependencies between methods.

Our research has led to the construction as depicted in Figure 7.1. The algorithm is started with an application of different preprocessing techniques to reduce the size of the problem. We then perform a simple lower bound calculation to get a lower bound quickly. At this point, the quality of the bound is not as important as the computing time. After this, the primal heuristic is used to try to find an upper bound. When knowledge of a lower- and upper bound is available, the row combination heuristic is applied to reduce the complexity of the problem.

Figure 7.1: The construction of LaRRS

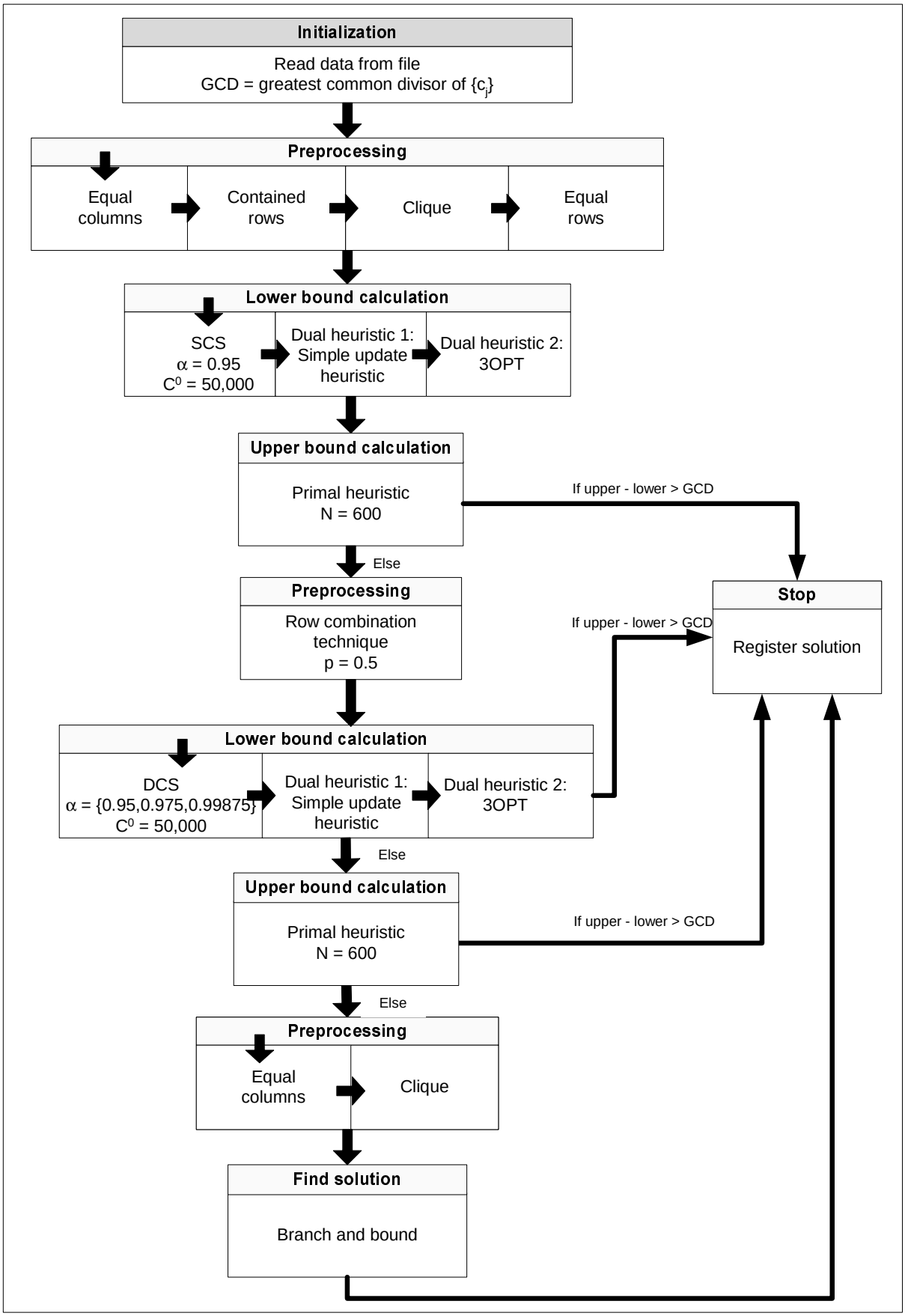


Table 7.1: Computational results of CPLEX and LaRSS on the test set

Name	Cols	Rows	Time LB CPLEX	Time CPLEX	Time LB LaRSS	Time LaRSS
nw41	197	17	0.020	0.030	0.016	0.016
nw32	294	19	0.010	0.060	0.016	0.016
nw40	404	19	0.030	0.050	0.015	0.015
nw08	434	24	0.010	0.030	0.030	0.030
nw15	467	31	0.030	0.110	0.000	0.000
nw21	577	25	0.030	0.050	0.000	0.000
nw22	619	23	0.030	0.050	0.016	0.047
nw12	626	27	0.030	0.050	0.016	0.031
nw39	677	25	0.030	0.060	0.031	0.031
nw20	685	22	0.020	0.060	0.047	0.047
nw23	711	19	0.020	0.060	0.031	0.031
nw37	770	19	0.030	0.050	0.016	0.016
nw26	771	23	0.030	0.050	0.000	0.000
nw10	853	24	0.030	0.050	0.047	0.047
nw34	899	20	0.030	0.060	0.016	0.016
Heart	926	180	0.080	405.410	0.578	0.625
nw43	1072	18	0.050	0.060	0.000	0.000
nw42	1079	23	0.030	0.060	0.060	0.060
Delta	1194	126	0.130	37.260	0.296	0.344
nw28	1210	18	0.030	0.080	0.000	0.000
nw25	1217	20	0.030	0.060	0.047	0.060
nw38	1220	23	0.050	0.080	0.031	0.031
nw27	1355	22	0.050	0.080	0.000	0.015
nw24	1366	19	0.030	0.060	0.016	0.016
nw35	1709	23	0.050	0.080	0.016	0.031
nw36	1783	20	0.050	0.250	0.078	0.109
Snowflake	2300	585	0.920	98.780	7.859	19.344
Fives	2440	72	0.080	0.560	1.032	1.093
Meteor	2464	60	0.060	5.450	0.234	0.266
nw29	2540	18	0.060	0.190	0.094	0.094
nw30	2653	26	0.050	0.140	0.031	0.063
nw31	2662	26	0.060	0.140	0.094	0.094
nw19	2879	40	0.060	0.130	0.015	0.031
nw33	3068	23	0.060	0.130	0.016	0.031
nw09	3103	40	0.060	0.110	0.281	0.313
nw07	5172	36	0.080	0.170	0.000	0.031
aa02	5198	531	0.640	0.690	0.781	1.187
nw06	6774	50	0.170	0.880	0.468	0.546
aa04	7195	426	1.270	50.720	2.438	> 24 hours
aa06	7292	646	1.280	3.720	1.766	5.875
kl01	7479	55	0.200	0.830	0.235	0.297
aa05	8308	801	2.250	4.160	1.735	6.078
aa03	8627	825	2.240	2.480	1.453	3.094
nw11	8820	39	0.140	0.380	0.296	1.109
aa01	8904	823	4.560	87.800	1.985	> 24 hours
nw18	10757	124	0.410	1.110	2.438	5.937
us02	13635	100	0.380	0.700	0.062	0.312
nw13	16043	51	0.270	1.000	0.735	0.968
us04	28016	163	0.730	1.280	0.297	0.672
kl02	36699	71	0.690	2.480	0.797	1.672
nw03	43749	59	0.880	3.480	1.907	2.578
nw01	51975	135	0.940	2.780	2.890	5.078
us03	85552	77	2.140	4.830	0.188	2.016
nw04	87482	36	1.580	39.890	4.547	6.453
nw02	87879	145	1.750	5.590	2.484	5.937
nw17	118607	61	2.310	16.750	1.827	3.266
nw14	123409	73	2.200	6.060	5.093	10.219
nw16	148633	139	7.890	16.980	3.874	12.765
nw05	288507	71	5.420	14.740	9.906	18.235
us01	1053137	145	259.660	389.830	11.750	56.485
Average	38585	123	5.041	20.154	1.184	2.996

After the row combination technique, a more elaborate subgradient search is performed in order to find a good lower bound, followed again by an application of the primal heuristic and some preprocessing techniques. Finally, the optimal solution is found by the branch and bound procedure. Every time we find a new lower - or upper bound, we check whether we have found the optimal solution and we apply the reduced cost fixing procedure of Section 3.5.1.

7.2 Computational results

This section examines the performance of LaRSS compared to the general purpose solver CPLEX 9.0 on our test set of 60 set partitioning problems. Within CPLEX, the default settings are used, with preprocessing turned on. Like most researchers in the field, we use CPLEX as our benchmark, since this is a well developed and widely available solver. The most successful algorithms from the literature, in particular Hoffman and Padberg (1993) and Borndörfer (1998) were not available to the author for comparison. Moreover, these solvers are several years old while CPLEX is improved continuously.

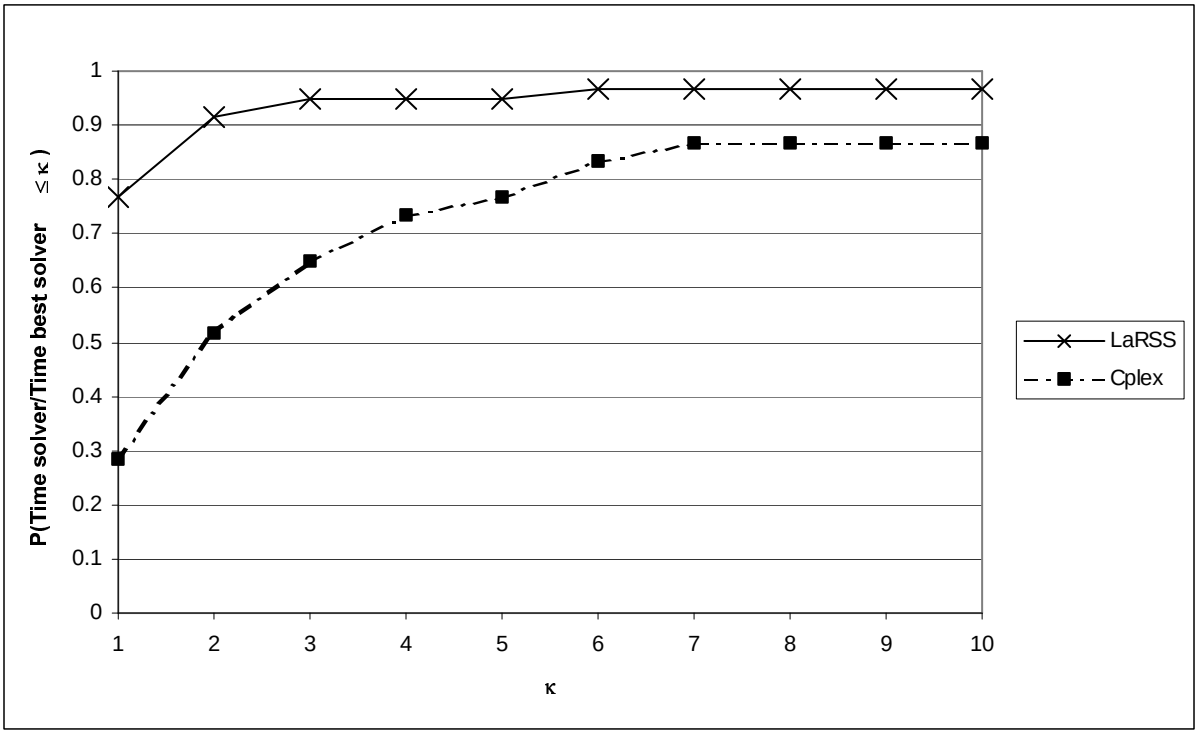
Table 7.1 shows the results of LaRSS and CPLEX on the 60 problems in the test set. Excluding aa01 and aa04, the total time of CPLEX on the 58 remaining instances is 1071 seconds, against 174 seconds for LaRSS. For 46 -or 79%- out of the 58 instances, LaRSS finds the optimum faster than or equally fast as CPLEX.

The two problematic instances, aa01 and aa04, are not solved within 24 hours with LaRSS. For aa01 no integer solution is found in this time period. The size of the partial solution during the branch and bound procedure, after the first 100,000 nodes, varies from 52 to 107 variables. For aa04, the best solution found is 29127, which is about 10% away from the optimal solution. The size of the solution during the branch and bound procedure varies from 24 to 57 columns. Section 5.5.1 considered our experience and the literature on these problems in more detail. Our experience corresponds to the findings of Borndörfer (1998). He observes that closing the gap from the dual side is what makes these problems difficult. This agrees with our experience; we have insufficient ability to improve the lower bound during the branch and bound procedure either with cuts, Lagrangian relaxation or dual heuristics. In Chapter 5 we found that aa04 indeed is solvable in less than an hour when we use additional lower bounding techniques during the branch and bound routine.

The relative performance of the two solvers on the test set of 60 instances is illustrated by the performance profile in Figure 7.2. The concept of performance profiles to compare optimization methods is discussed in Dolan and Moré (2002). The figure indicates for both solvers the probability, based on the current test set, that the computing time of the solver is within κ times the time of the best solver. The profile shows that LaRSS is the best performing solver for 77% of the 60 problems in the test set, while CPLEX is the best performing solver for 28% of the problems.

Moreover, it shows that the computing time of LaRSS is within a factor three of the time of the best solver for 95% of the problems. On the other hand, the solution time of CPLEX is within a factor three of the time of the best solver for about 65% of the problems. For 58 out of the 60 problems -or 97%- the computing time of LaRSS is within a factor six of the best time. The computing time of CPLEX is within a factor six of the best time for only 83% of the problems. The performance profiles indicate that LaRSS performs well compared to CPLEX, since the performance line of LaRSS is above the line of CPLEX for κ smaller than or equal to ten.

Figure 7.2: Performance profile of LaRSS and CPLEX on the test set



7.3 Technical aspects

No matter how well developed the theoretical aspects of a solver are, its performance ultimately depends on the implementation of the techniques. Two technical factors contribute greatly to this performance: efficient implementation of developed techniques and management of the data. This section gives some insight into, and examples of, both aspects with respect to LaRSS.

7.3.1 Efficiency

The code of LaRSS is entirely written in C. Many books have been written on efficient programming in C and C++. We will therefore not elaborate on the programming tips

and tricks that can improve the performance of C and/or C++ code. For a reference, see Bulka en Mayhew (2000) and Zaratian (1999). We will only comment on some efficiency related issues considering the methods implemented in LaRSS. Although we will not discuss the implementation of all methods, we will provide some examples of how implementation aspects can influence the efficiency of an algorithm. Note that in every software application, efficiency depends heavily on the implementation. Although there is much literature on standard software design and standard algorithms (see, for example, Press et al., 2002), the issues examined here are specific to the LaRSS solver and are typically discovered by simple trial-and-error.

Example 1

Consider the equal columns rule discussed in Chapter 2. This preprocessing rule can be implemented in different ways. The simplest implementation would just check every pair of columns and see whether the nonzero elements are equal. However, since we have information about the number of nonzero elements per column, we can use this information and only check those pairs of columns for which these numbers are equal. This would still result in a high number of detailed checks of pairs of columns. Therefore, we add another step, and calculate for each column the sum of squares of the nonzero elements covered by this column. The implementation of the total rule is given in Appendix A. To illustrate the effect of such a seemingly simple additional procedure, we compare the results of:

1. Sorting the columns by the number of nonzero elements; checking pairs of columns with the same number of elements
2. Sorting the columns by the sum of squared indices of nonzero elements; checking pairs of columns only if this number is equal

Table 7.2: Computing times for the first efficiency example

	Method 1	Method 2
Total time	6072.92	6.88
Time nw03	9.11	0.06
Time nw01	8.33	0.14
Time us03	22.84	0.45
Time nw04	33.06	0.17
Time nw02	33.38	0.17
Time nw17	71.69	0.33
Time nw14	89.88	0.33
Time nw16	119.89	0.34
Time nw05	586.34	0.89
Time us01	5078.00	3.81

Table 7.2 shows the total computing time of these procedures on our test set, as well as the separate computing times for the ten largest instances. The results clearly indicate the value of the pretest in the equal columns preprocessing rule.

Example 2

Consider the dynamic convergent series we introduced in Chapter 3 to solve the Lagrangian relaxation problem. In this method, as in every other subgradient search method, the Lagrangian multipliers are adjusted iteratively and the Lagrangian costs of the columns are updated accordingly. Suppose that we are at a certain iteration of this procedure, where we have adjusted the Lagrangian multiplier for a number of rows. We know that a certain number of columns and rows are not active due to the preprocessing and reduced cost-fixing techniques. We consider two different approaches to adjust the Lagrangian costs of the columns, illustrated in Figure 7.3.

Figure 7.3: Two procedures for updating Lagrangian costs during subgradient search

Procedure 1: Column-wise updating	Procedure 2: Row-wise updating

FOR($j \in J$)
  IF(NOT(j deleted))
  FOR($r \in R(j)$)
  IF(NOT(r deleted) AND (λ_r changed))
  $cl_j = cl_j$ – change_in_ λ_r
  ENDIF
  ENDFOR
  ENDIF
ENDFOR
	FOR($r \in R$)
	  IF(NOT(r deleted) AND (λ_r changed))
	  FOR($j \in J(r)$)
	  IF(NOT(j deleted))
	  $cl_j = cl_j$ – change_in_ λ_r
	  ENDIF
	  ENDFOR
	  ENDIF
	ENDFOR

To illustrate the difference between these two approaches, we have applied the dynamic convergent series method on our test set, using both procedures. Table 7.3 shows the total time of this method for both cases, as well as the individual computing times of the ten largest instances in the test set. Clearly, the first method can be carried out much faster. Generally, the number of columns of an SPP is much higher relative to the number of rows, such that it is more efficient to view only those columns for which we know for sure that the costs have to be adjusted.

Table 7.3: Computing times for the second efficiency example

	Procedure 1	Procedure 2
Total time	101.59	354.77
Minimal time	0.03	0.06
Maximal time	21.58	81.38
Time nw03	2.56	7.63
Time nw01	2.50	15.78
Time us03	1.78	8.39
Time nw04	8.88	30.95
Time nw02	1.64	10.97
Time nw17	2.13	7.00
Time nw14	5.52	19.83
Time nw16	21.58	81.38
Time nw05	11.27	46.88
Time us01	13.45	68.92

7.3.2 Data management

A great deal of data must be maintained during the different procedures applied within LaRSS. For every column j, we need to preserve the following:

- Original index
- Original costs c_j
- Reduced costs cr_j
- Number of nonzero elements, |R(j)|
- The actual nonzero elements, R(j)
- Whether column j is deleted or not

For every row r we need to preserve the following:

- Dual value u_r
- Number of nonzero elements, |J(r)|
- The actual nonzero elements, J(r)
- Whether row r is deleted or not

There are two different ways to store these data: in arrays or in records. In the array structure, we need six arrays to store the data of the columns, where every array covers one characteristic. When the columns are sorted, we have to sort each of these arrays. In the record structure, the columns are stored in an array of records, where every record covers one column and contains all the characteristics of that particular column. When columns are sorted, we now only have to sort the array of records and all characteristics are sorted accordingly. The advantage of the record structure obviously lies in the speed of the sorting procedures and the convenience of

use. In most cases, however, elementary operations are performed more rapidly with the array structure. We illustrate these claims with two examples.

Example 1

Consider the equal columns preprocessing rule discussed in Chapter 2. This preprocessing rule involves sorting of the columns, and since the equal columns rule is performed first in LaRSS, the total set of columns must be sorted. We therefore expect that performing this procedure on the columns in the record structure is faster than when we perform it on the columns in the array structure. Table 7.4 shows the total time of the equal columns procedure method for both data structures, as well as the individual computing times of the ten largest instances in the test set. Indeed the procedure is performed faster in the record structure, the difference being 30%.

Table 7.4: Computing times for the first data management example

	Array	Record
Total time	6.88	4.78
Minimal time	0.00	0.00
Maximal time	3.81	2.56
Time nw03	0.06	0.06
Time nw01	0.14	0.08
Time us03	0.45	0.19
Time nw04	0.17	0.16
Time nw02	0.17	0.14
Time nw17	0.33	0.22
Time nw14	0.33	0.22
Time nw16	0.34	0.27
Time nw05	0.89	0.55
Time us01	3.81	2.56

Example 2

Consider the primal heuristic discussed in Chapter 4. Table 7.5 shows the total time of the primal heuristic for both data structures, as well as the individual computing times of the ten largest instances in the test set. The primal heuristic is performed faster within the array data structure, the difference being 16%.

These two examples illustrate that neither of the two data management systems performs better than the other in all cases. In LaRSS, we use array structures to store the characteristics of the rows as well as the columns.

Table 7.5: Computing times for the second data management example

	Array	Record
Total time	5.58	6.47
Minimal time	0.00	0.00
Maximal time	2.38	2.42
Time nw03	0.06	0.09
Time nw01	0.22	0.30
Time us03	0.09	0.09
Time nw04	0.92	1.13
Time nw02	0.16	0.19
Time nw17	0.38	0.47
Time nw14	0.14	0.19
Time nw16	0.36	0.42
Time nw05	0.34	0.38
Time us01	2.38	2.42

7.4 Concluding remarks

This chapter examined the Lagrangian relaxation-based set partitioning solver LaRSS, which is constructed from the techniques discussed in this thesis. We considered the construction of the solver, the computational results and some technical aspects.

Apart from two difficult cases, LaRSS performs very well: the total time of LaRSS on the 58 instances is only 174 seconds against 1,070 for CPLEX. Including the two difficult cases, LaRSS performs better on 77% of the problems. For 97% of the problems, the computing time of LaRSS is within a factor six of the best time of the two solvers, while for CPLEX this is the case for only 83% of the problems. The performance profiles of LaRSS and CPLEX indicate that LaRSS has a better overall performance on the test set.

The two difficult cases, however, are not solved by LaRSS within 24 hours, while CPLEX solves them within minutes. The difficulty of these instances is recognized in the literature on set partitioning problems, while a cause for this phenomenon has never been established. It seems that the problems with these instances come from the dual side, where we have an obvious disadvantage compared to linear programming based algorithms, since using lower bounds and cuts during branch and bound is much easier when information about the linear programming basis and solution is available.

Chapter 8

Case study: collection of liquids coming from end-of-life vehicles

This chapter deals with a case study performed by CentER Applied Research for Auto Recycling Nederland, using LaRSS to solve set partitioning problems. The content of this chapter appeared in revised form as Le Blanc et al. (2004A).

Section 8.1 we discusses the setting of the case study. Section 8.2 considers the literature on similar problems. Section 8.3 desribes the methodology and the corresponding model. Section 8.4 discusses the results of the case study. Section 8.5 examines the statistics considering the set partitioning problems in three different scenarios. Finally, Section 8.6 provides some concluding remarks.

8.1 Introduction

In 1993, the Dutch automotive industry founded an organization for the recycling of end-of-life vehicles (ELVs), named Auto Recycling Nederland (ARN). Consumers can turn in their car for free at an ARN-certified dismantler, regardless of the brand of the car. This system is financed by a fee that is charged to buyers of new cars. This way of organizing end-of-life recycling is very common in the Netherlands and similar branch organizations exist for several other end-of-life streams, for example white-and-brown goods (De Koster et al., 2005), batteries and tires.

In the European directive (Directive 2000/53/EC), the European Union prescribes guidelines for the legislation on recycling of ELVs in EU member states. In 2002, this EU directive was implemented by all member states. The Dutch legislation prescribes that wrecks should be recycled and reused for at least 85% of the average vehicle weight.

While vehicles are waiting for removal of hazardous materials, they must be

stored at a location with an impermeable floor, in order to prevent environmental damage. The old drainage methods did not meet the latest requirements on safety and the environment, because liquids, for example, were often accidentally spilled. The installation of a new drainage system at 265 ELV dismantler sites affiliated to ARN started in 2003. With the new equipment, liquids are siphoned off ELVs to a storage reservoir in a closed system without any chance of spilling. A large storage vessel is installed for each liquid, equipped with remote monitoring. There are vessels for fuel, oil, coolant and windscreen washer fluid.

At the time of the research, collection of these liquids took place when the collection company received a message from the waste generator that a reservoir was almost full. Using data from the remote monitoring equipment (telemetry), one can foresee this several weeks ahead. This information is valuable and should be exploited. We have developed a new procedure where the collection company is responsible for timely collection. In this, what we call, collector managed inventory (CMI) situation, the collection company periodically (e.g. weekly) retrieves data on the inventory levels of the storage vessels and constructs a collection plan. The tanker trucks used for collection consist of two compartments for different liquids. The two possible combinations are oil and coolant or fuel and windscreen washer fluid. If the data that stem from the telemetry units indicate that one of the materials needs to be collected, then both materials are collected at the same time. Materials collected together are called material groups; not all materials can be collected by one truck. Collections can take place for two reasons:

- They can be volume driven: the storage reservoirs are almost full and collection is needed to prevent capacity shortages.
- They can be time driven: there is a minimal collection frequency that should be respected.

Minimal collection frequencies can be used for materials for which the quality deteriorates over time. For example, oils are hygroscopic (attract water); they should therefore be collected at least once a year to assure sufficient quality for recycling.

8.2 Literature

Collector Managed Inventory (CMI) can be seen as the reverse logistics counterpart of the well-known concept of Vendor Managed Inventory (VMI) in forward logistics. In a VMI system, the supplier decides on the appropriate inventory levels of each of the products and the appropriate policies to maintain these levels (Simchi-Levi et al., 2000). To the best of our knowledge, CMI has never been introduced in the reverse logistics literature as the counterpart of VMI. While the idea of monitoring the level of refuse or recyclables collected and dynamically scheduling the collection as an alternative for periodic collection systems is mentioned in Beullens (2001), the practical implementation of a CMI system has never been investigated.

In the literature on forward logistics, a number of papers consider concepts with similar characteristics as discussed here. These papers address so-called inventory routing problems. Inventory routing problems involve a set of customers with a certain daily demand to be served from a central depot. The objective is both to minimize costs and to prevent customers from running out of stock (Dror and Ball, 1987).

One of the first papers to address the inventory routing problem was Bell et al. (1983), describing a project for forecasting inventory levels at an industrial gas supplier. To avoid shortages in the long-term, minimum levels on the inventory of customers at the end of the planning period are defined. Based on forecast information, the actual scheduling process is solved by a mixed integer linear programming model. The programming model selects from the total set of possibilities the best subset of routes to be driven and the amount to be delivered to the customers. The set of possible routes is limited because of the small number of customers on a trip and many practical restrictions on the routes. All logical routing possibilities are enumerated explicitly and fed into the programming model.

All inventory routing models have to incorporate the long-term effects of decisions taken in the current operational planning period. Dror and Ball (1987) reduce the long-term horizon by considering penalty costs expressing the long-term effects of decisions made in the operational planning period. Only full replenishment of inventories is considered. The resulting planning problem is solved in a three-phase approach. In phase 1, the customers are assigned to days. In phase 2, the vehicle routing problem is solved using a Clarke and Wright savings algorithm (Clarke and Wright, 1964). In phase 3, the solution obtained for phase 2 is improved by considering exchanges. Dror and Levy (1986) explain how these improvement methods work for inventory routing.

Herer and Levy (1997) notice the disregard of an appropriate treatment of inventory holding costs in the above literature. They model the problem by incorporating inventory holding costs by so-called *temporal distances*. Customers that are spatially close tend to be on the same route if they are also temporally close, meaning that the optimal delivery periods are not too far apart. The effects of short-term decisions on the long-term holding, shortage and fixed ordering costs are incorporated in the temporal distances. Temporal distances are defined as the minimal costs of combining two customers in a common delivery period. The temporal distances are used in the savings calculation of the Clarke and Wright algorithm (Clarke and Wright, 1964).

Campbell et al. (2002) describe an application in the distribution of industrial gases. The planning is solved with a rolling horizon in a two-phase approach. In the first step, an integer program is used to determine which customers will be visited and how much will be delivered. Clustering and aggregation techniques are used to make the integer program solvable. In the second step, an insertion heuristic combined with several improvement heuristics is used to determine the actual

delivery routes. Inventory holding costs are not considered.

In essence, the problem setting described above is similar to the reverse logistics setting as described in this chapter. Instead of delivering gases or soft drinks, one delivers storage space for oils and fuels. Nevertheless, the setting of ARN has some characteristics that are different and justify a new model. Due to the low or sometimes even negative value of the cores or materials to recycle, the inventory holding costs are irrelevant. Collecting as much as possible in one visit is the best approach, thereby minimizing transportation costs. Since inventory costs do not matter and the supply rate of the liquids is low, the time between two consecutive visits is long in contrast to the applications described in the literature discussed above. The demand for the goods and gases in these cases is unknown and sometimes difficult to estimate; consider, for example, the demand for heating oil, which depends on the weather. In our problem setting, we have the opportunity to obtain accurate information on the levels of fluids at the waste generators, due to the online monitoring of the level of the reservoirs.

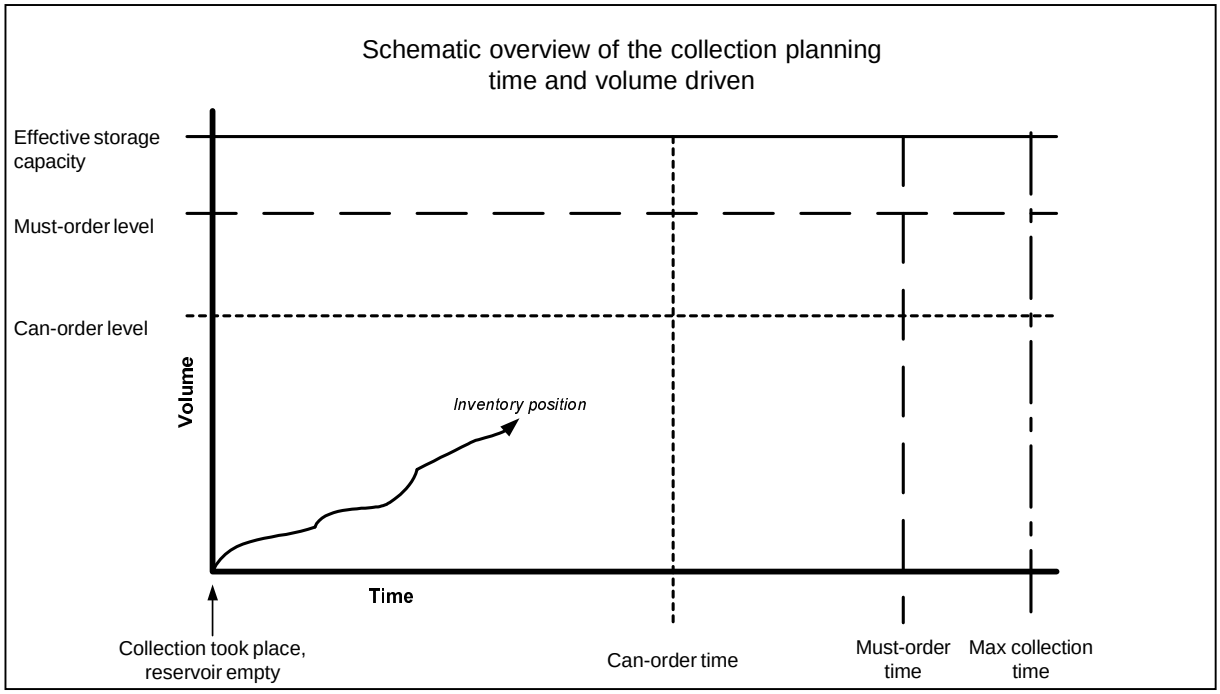
8.3 Model

8.3.1 Planning methodology

The planning is of a periodic nature. We assume a periodic review of inventory levels. After retrieval of the data with the telemetry units, a collection plan is constructed for the coming review period. An order triggered either by volume or by time, must be performed in the coming planning period. We refer to these orders as must-orders. We also consider the possibility of visiting dismantlers that do not directly need a collection but are close to triggering one and can be inserted in the route at low marginal costs. These orders are called can-orders. Can-orders are used to profitably fill up the remaining capacity of collection trucks but can never cause a new collection trip. An additional difference is that can-orders can be performed partially, meaning not fully emptying the storage reservoir but only as far as capacity left in the truck allows, while must-orders must fully empty the storage reservoir.

Figure 8.1 shows a conceptual overview of the must-order level, can-order level, the must-order time and the can-order time for a given storage reservoir at a dismantlers site. Inventory levels are monitored at the beginning of each collection period, equal to one week in the base scenario. When a dismantler passes one of the must-order lines, a must-order is generated that will be part of a route. If a can-order line is passed, but not a must-order line, then a can-order is generated for possible insertion in a must-driven route. If one of the materials in a material group triggers an order, the order is generated for the whole material group.

Figure 8.1: Conceptual overview of collection planning



The must-order level is defined analogously to the reorder-point in inventory management theory, see e.g. Silver et al. (1998). Assume that the material collection for material *mat* of dismantler *ed* is normally distributed with mean $\mu_{ed,mat}$ and variance $\sigma^2_{ed,mat}$. The effective storage capacity for material *mat* of dismantler *ed* is given by $cap_{ed,mat}$. The length of the planning period or review period is denoted by *rp*. The collection takes place within the planning period, so the response time is at most *rp* days; we assume that the response time is uniformly distributed.
The must-order level, $mo\_level_{ed,mat}$ for dismantler *ed* and material *mat* is given by:

$$mo\_level_{ed,mat} = cap_{ed,mat} - 1\tfrac{1}{2} \cdot rp \cdot \mu_{ed,mat} - k_{ed,mat} \cdot \sqrt{rp \cdot \left(1\tfrac{1}{2} \cdot \sigma^2 + \tfrac{1}{12} \cdot \mu^2 \cdot rp\right)} \qquad [8.1]$$

The safety factor $k_{ed,mat}$ is used to capture the uncertainty within the collection period. The safety factor can easily be calculated using the standard normal distribution and the desired service level. The can-order level $co\text{-}level_{ed,mat}$ is given by:

$$co\_level_{ed,mat} = mo\_level_{ed,mat} - \alpha_{ed,mat} \cdot rp \cdot \mu_{ed,mat} \qquad [8.2]$$

Here, $\alpha_{ed,mat} \in \{0,1,2,\ldots\}$ expresses the number of planning periods that we look forward for can-orders driven by volume. When the parameter $\alpha_{ed,mat}$ equals zero for all dismantlers and materials, this corresponds to a policy without can-orders. The must-order time, $mo\_time_{ed,mat}$, is based on the maximum allowed number of days between two collections, max collection time, resulting in:

$$mo\_time_{ed,mat} = \max \text{collection time} - rp \qquad [8.3]$$

The can-order time is calculated using $\beta_{ed,mat} \in \{0,1,2,\ldots\}$, expressing the number of planning periods we look forward for can-orders driven by time:

$$co\_time_{ed,mat} = mo\_time_{ed,mat} - \beta_{ed,mat} \cdot rp \qquad [8.4]$$

At the beginning of each collection period, the inventory positions are retrieved for all storage reservoirs for all dismantlers. This information is used to generate the must-orders and can-orders and to create a collection plan for the coming period. This plan is constructed by generation of feasible routes and selecting the optimal combination of routes by solving a set partitioning problem.

8.3.2 Route generation

In the route generation procedure, every possible route is generated. If the route is found feasible, it is written to the set partitioning tableau. A route is feasible if the maximum time allowed for one day, or the maximum capacity of one of the truck reservoirs, is not exceeded. Since the number of orders considered per period is relatively small and the number of orders that fit in a route is limited, explicit enumeration of all possible routes is possible. The difficulty in route generation is enumerating all combinations in a systematic and efficient way. Our route generator consists of two main procedures that are used recursively: *MustOrderInsertor* and *CanOrderInsertor*. These procedures aim to add an unplanned must- or can-order, respectively, to the route. If a route is found feasible, it is written to the set partitioning tableau and an attempt is made to add another order. If a route is found to be infeasible, the order added last is removed from the route and a new attempt is made to add the next order in the list. Figure 8.2 provides an overview of these two main procedures. The route generation process starts with an empty route and a call to the procedure *MustOrderInsertor*.

During the route generating process, the costs of the route are calculated and corrected for the costs of inserting can-orders, the future savings. These cost savings, as shown in equation [8.5], are based on the difference between the costs of insertion in the current route and the costs of a separate route for this order (linehaul), corrected for the amount of material. This savings mechanism evaluates the benefit of adding a can-order to the existing route, compared to waiting until collection is necessary, i.e. a must-order is generated. It acts as a selection mechanism for can-orders.

$$\text{cost\_savings}_{ed} = \text{insertion\_cost}_{ed} - \frac{\text{linehaul\_cost}_{ed}}{\sum_{mat} \text{linehaul\_volume}_{ed,mat}} \cdot \left(\sum_{mat} \text{inserted\_volume}_{ed,mat} \right) \qquad [8.5]$$

Example

Consider an ELV-dismantler having a maximum storage capacity for 3,000 liters of oil and 2,000 liters of coolant. The costs of collecting these materials in a linehaul, when both vessels are full, is equal to € 200, i.e. € 0.04 per liter. At a given moment, this ELV-dismantler can be inserted in a route as a can-order at an insertion costs of € 90. The total quantity that can be collected of both oil and coolant is 4,000 liters. The

estimated cost savings of inserting this can-order using formula [8.5] are € 70, compared to waiting until collection is necessary, i.e. a must-order is generated.

The same cost savings factor is used for must-orders in non-empty routes, because combining must-orders as much as possible in a route reduces the total number of routes to be driven. This correction for must-orders is necessary, since otherwise driving two routes with one must-order in each route, combined with a number of can-orders, would be evaluated better than driving one route with both must-orders.

Figure 8.2: Outline of the route generator

```
Function MustOrderInsertor
FOR (MOrder in UnplannedMustOrderList) DO
        RouteFeasible = MustOrderInsertInRoute(MOrder)
        Remove MOrder from UnplannedMustOrderList
        IF(RouteFeasible)
                WriteRouteToSP
                MustOrderInsertor
                CanOrderInsertor
        ENDIF
        MustOrderRemoveFromRoute(MOrder)
        Add MOrder to UnplannedMustOrderList
ENDFOR

Function CanOrderInsertor
FOR (COrder in UnplannedCanOrderList) DO
        RouteFeasible = CanOrderInsertInRoute(COrder)
        Remove COrder from UnplannedCanOrderList
        IF(RouteFeasible) THEN
                WriteRouteToSP
                CanOrderInsertor
                ENDIF
        CanOrderRemoveFromRoute(COrder)
        Add COrder to UnplannedCanOrderList
ENDFOR
```

8.3.3 The route selection problem

We formulate the problem of finding a collection of routes such that all must-orders are fulfilled with minimal costs as an integer programming problem. After the introduction of some notation, this problem is given in equations [8.6] to [8.12] below.

Variables

$X_{r,vd}$ = 1 if route r is chosen for vehicle-day combination vd; 0 otherwise.

sc_{co} = 1 if can-order co is not fulfilled in the chosen routes; 0 otherwise.

sv_{vd} = 1 if vehicle-day combination vd is not used to fulfill the chosen routes; 0 otherwise.

Parameters

c_r = the costs of route r, corrected for can-orders and multiple must-orders in the route.

$a_{mo,r}$ = 1 if must-order mo in route r; 0 otherwise.

$a_{co,r}$ = 1 if can-order co in route r; 0 otherwise.

The route selection problem

$$\min \quad \sum_r \sum_{vd} c_r \cdot X_{r,vd} \qquad [8.6]$$

Subject to

$$\sum_r \sum_{vd} a_{mo,r} \cdot X_{r,vd} = 1 \qquad \forall mo \qquad [8.7]$$

$$\sum_r \sum_{vd} a_{co,r} \cdot X_{r,vd} + sc_{co} = 1 \qquad \forall co \qquad [8.8]$$

$$\sum_r X_{r,vd} + sv_{vd} = 1 \qquad \forall vd \qquad [8.9]$$

$$X_{r,vd} \in \{0,1\} \qquad \forall r,vd \qquad [8.10]$$

$$sc_{co} \in \{0,1\} \qquad \forall co \qquad [8.11]$$

$$sv_{vd} \in \{0,1\} \qquad \forall vd \qquad [8.12]$$

Equation [8.6] describes the objective function of the optimization problem, which is of course total cost minimization of the collection plan. Equation [8.7] represents the constraints assuring that each must-order is executed exactly once. Equation [8.8] represents the constraints assuring that each can-order is inserted at most once. Equation [8.9] assures that each vehicle-day combination has at most one route. Equations [8.10] - [8.12] bound the domain of the variables. The variables sc_{co} and sv_{vd} in constraints in [8.8] and [8.9] serve as slack variables to rewrite the problem to a pure set partitioning problem.

In many instances, the number of vehicle days available exceeds the number of must-orders. In these cases the vehicle-day combination becomes irrelevant, since there will never be more routes than must-orders. When this is the case, we can skip the index ‘vd’ and the number of variables can be reduced with a factor equal to the number of vehicle-day combinations.

In some cases, the generated set-partitioning problem is infeasible, because the available capacity (vehicle-day combinations) is too small to fulfill all the must-orders. To overcome this, we have added a dummy route for each must-order. This dummy route covers only one must-order and the costs of this route are equal to a certain factor times the costs of a linehaul. These costs represent the costs of an emergency order and assure that the dummy route will be chosen only if it is not possible to fulfill the order on a vehicle-day combination. The set partitioning problem given by equations [8.6] – [8.12] is solved with LaRSS. Section 8.5 discusses the statistics considering the sizes and computing times of the several scenarios.

8.3.4 Validation and verification

We performed extensive tests on the model to validate and verify its correctness. In the verification process we analyzed the internal consistency of the model, in particular by pushing the parameters to the extremes of the spectrum. In this way, the behavior and outcome of the models are checked on logic. In the validation process we have tested the external correctness of the model: does the model give representative descriptions of the real world system? We compared the results with data coming from collection companies. Furthermore, logistic specialists of ARN examined the model outcomes, comparing them with their expectations.

8.4 Case results

8.4.1 Scenario data

We simulate a horizon of ten years. Since, the base scenario uses a collection period of a week, a total of 522 collection periods are simulated. Collection should take place at least once a year; all ELV-dismantlers are thus visited at least ten times in the simulation. The initial inventory at the first collection period for each of the 267 ELV-dismantlers was generated randomly. We use the same initial situation in order to make a fair comparison for each scenario.

For the collection of oil and coolant, a tanker truck with a capacity of 7,600 liters for oil and 5,700 liters for coolant is rented. The collection company rents these tanker trucks, including the driver, to different customers; ARN is therefore only charged for the usage, expressed in the number of hours and kilometers driven. A regular workday consists of 450 minutes, after that the charge per hour is doubled for the next 240 minutes. The starting and unloading point is the current depot for oil and coolant, located in Lelystad, in the Netherlands.

8.4.2 Base scenario with partial and full collection of can-orders

The situation with reactive planning coincides with the situation in which can-order level $\alpha_{ed,mat}$ and can-order time $\beta_{ed,mat}$ are both 0, see equations [8.2] and [8.4]. A proactive approach coincides with can-order level $\alpha_{ed,mat}$ and can-order time $\beta_{ed,mat}$ larger than 0. We varied $\alpha_{ed,mat}$ and $\beta_{ed,mat}$ between 0 and 6, where we did not differentiate in ELV-dismantlers or materials. The results for different can-levels and can-times are shown in Table 8.1.

We observe that a possible cost reduction up to 18.9% is realized by adopting a forward-looking strategy. The number of routes necessary for collection, which is

equal to the number of vehicle days, as well as the average distance traveled within each route, is reduced. The total number of kilometers driven per year is reduced by about 18,700. Consequently, it is no surprise that the load-factor, the maximum fraction of capacity of the truck used in a route, is increased from 0.67 to 0.93.

Table 8.1: Results for oil and coolant with fractional collection of can-orders

α (can-order level)	0	1	2	3	4	5	6
β (can-order time)	0	1	2	3	4	5	6
Average # must-orders per week	7.17	5.50	4.98	4.66	4.53	4.45	4.43
Average # can-orders per week	0.00	2.13	2.80	3.28	3.47	3.61	3.67
Average # routes per week	3.95	3.41	3.34	3.27	3.26	3.23	3.23
Average route distance (km)	345.1	337.7	326.2	320.6	315.0	313.0	310.9
Average route duration (min)	541	562	556	556	551	552	550
Average load-factor	0.671	0.840	0.883	0.909	0.923	0.927	0.931
Kilometers driven per year	71,091	60,178	56,824	54,758	53,613	52,709	52,387
Costs per year (normalized)	100	89	86	84	82	82	81

In the base scenario described above, we assumed that partial execution of can-orders is allowed. Table 8.2 gives the results of a scenario in which we restrict the model to allow only full collection of can-orders. In this scenario, a significant cost reduction of 10.5% opposed to 18.9% is possible when we adopt the same proactive strategy. The load-factor is increased up to 0.80, which is significantly less than in the situation where we allow fractional can-orders.

Table 8.2: Results for oil and coolant with full collection of can-orders

α (can-order level)	0	1	2	3	4	5	6
β (can-order time)	0	1	2	3	4	5	6
Average # must-orders per week	7.17	6.20	5.64	5.38	5.22	5.08	4.98
Average # can-orders per week	0.00	1.00	1.63	1.98	2.26	2.47	2.65
Average # routes per week	3.95	3.69	3.58	3.54	3.45	3.42	3.39
Average route distance (km)	345.1	348.7	348.1	342.2	344.6	341.6	337.0
Average route duration (min)	541	558	563	560	568	568	566
Average load-factor	0.671	0.707	0.733	0.754	0.773	0.781	0.800
Kilometers driven per year	71,091	67,090	65,025	63,239	62,062	60,976	59,582
Costs per year (normalized)	100	97	95	93	92	91	89

Figures 8.3 and 8.4 illustrate the collection costs and load-factor for both scenarios; the intervals in both figures indicate the 90% confidence intervals. The figures illustrate the decrease of the marginal cost savings by extending the forward-

looking horizon. A forward-looking period of three weeks ($\alpha_{ed,mat} = \beta_{ed,mat} = 3$) seems to be enough to fully exploit the savings potential.

Figure 8.3: Yearly collection costs with fractional and full collection of can-orders

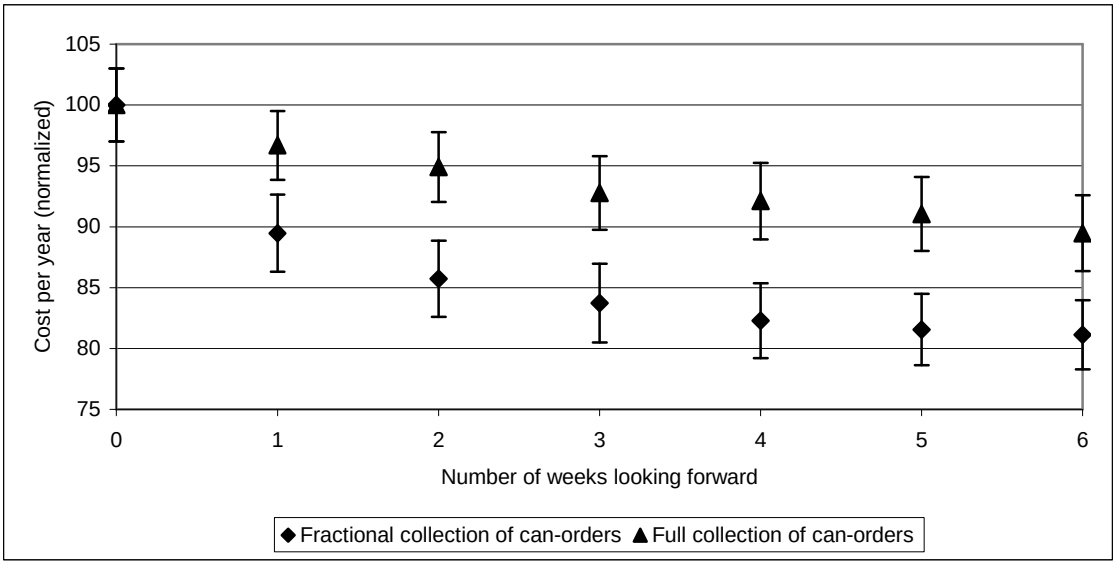
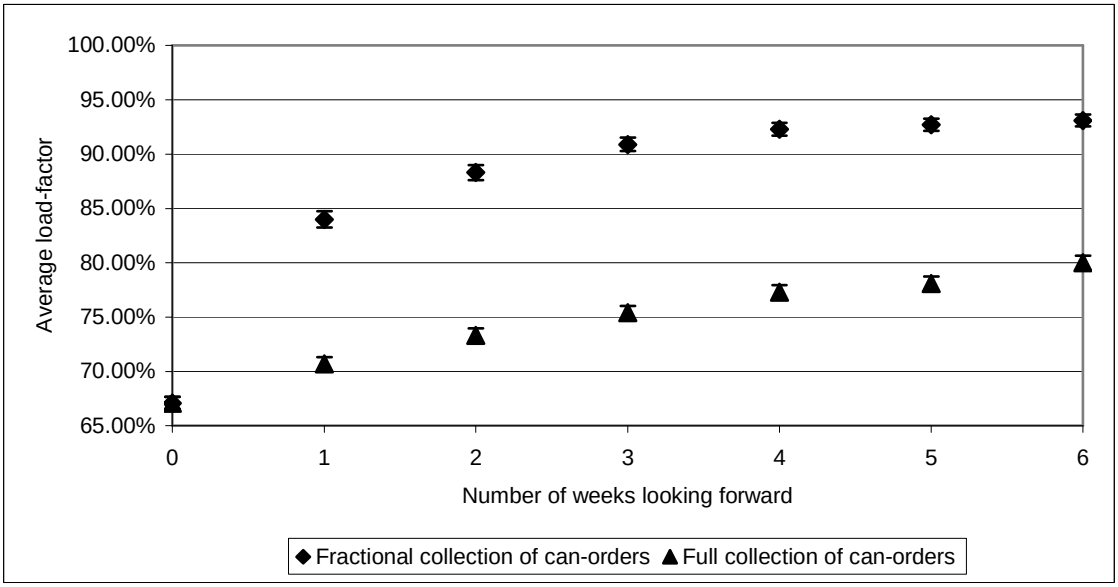


Figure 8.4: Average load-factor with fractional and full collection of can-orders



8.4.3 Sensitivity analysis on the length of collection period

The choice for the review period of one week is a somewhat arbitrary management decision. If this frequency is increased, we expect that the performance will increase as well, since we can plan the trips more frequently, using more up-to-date data. However, when the length of the planning period is larger, we have more possibilities to combine orders and to create more efficient routes. This is illustrated by Figures

8.5 and 8.6, which depict the costs and the load factor for different lengths of the collection period. In the reactive strategy, longer collection periods perform better, which is a result of more combination possibilities. However, when we adopt a proactive strategy, we already have better combination possibilities by looking forward. In summary: the more proactive, the higher the planning frequency should be. However, the relative improvement of changing the planning frequency is small compared to the shift from reactive to proactive planning. Since collection periods of one week are more convenient, this justifies the management decision with a proactive strategy.

Figure 8.5: Yearly collection costs for different lengths of the collection period

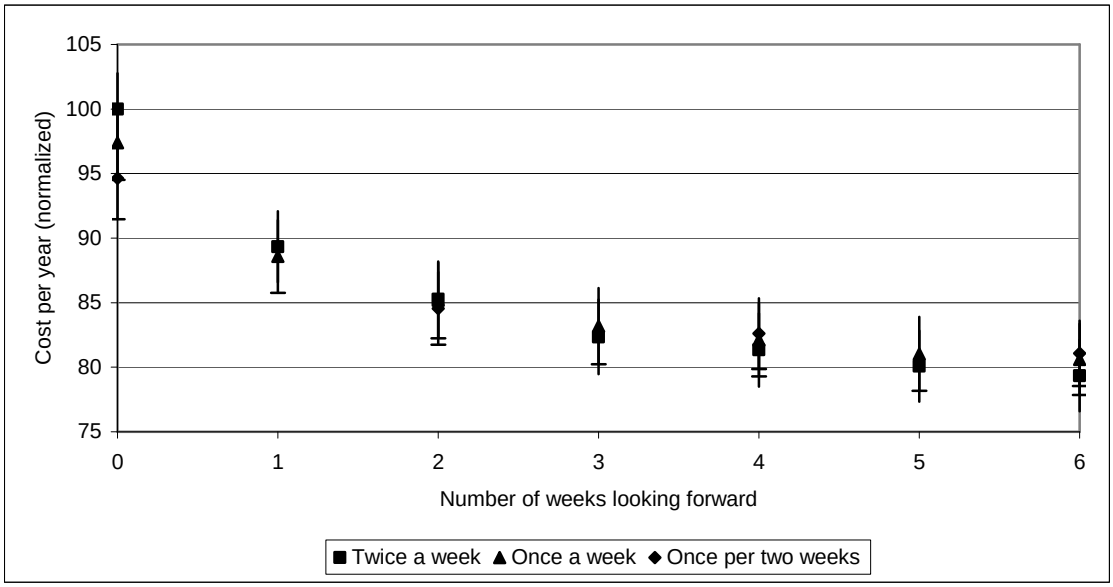
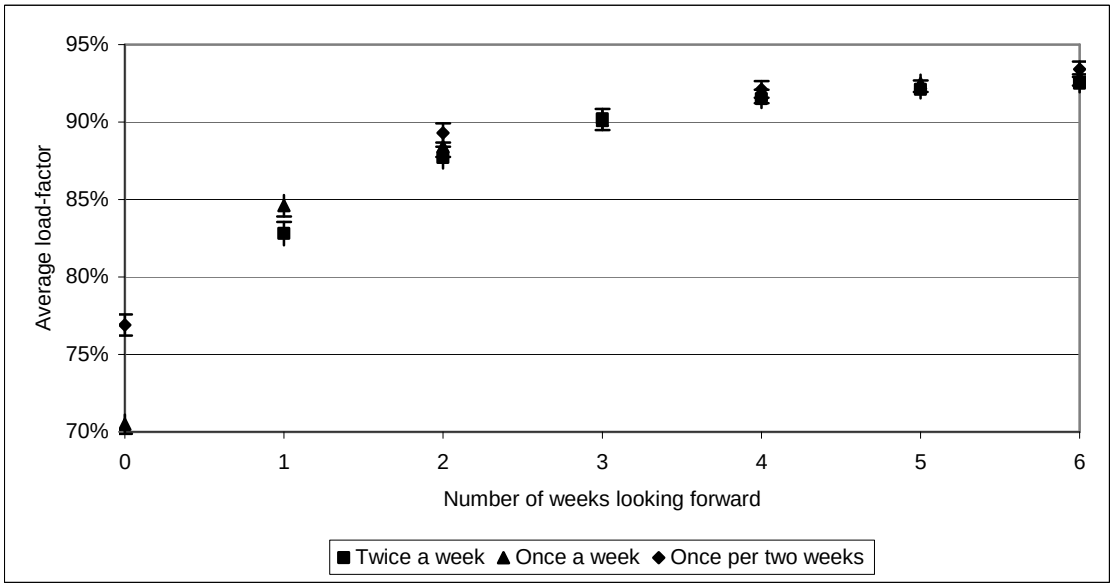


Figure 8.6: Average load-factor for different lengths of the collection period



8.5 Set partitioning results

In the analysis for Auto Recycling Nederland, several scenario calculations were performed to investigate the influence of the parameters of the system. We investigate the performance of LaRSS on the instances belonging to three different scenarios:

1. Base scenario
2. Alternative scenario in which trucks have double capacity.
3. Alternative scenario in which the review period is three times as long

The computational experiments discussed in this section are performed on a normal desktop computer, running on MS Windows 2000 with a 1000 MHz Pentium processor and 512 MB RAM.

8.5.1 Base scenario

The first scenarios we examine are the base scenarios that are described in Tables 8.1 and 8.2, with full and fractional collection of can-orders, respectively. Tables 8.3 and 8.4 show the statistics of the set partitioning problems for these base scenarios with full and fractional collection of can-orders, respectively. Note that when $\alpha = \beta = 0$ there is no proactive planning and no can-orders are triggered. In this case the planning problems for full and fractional collection are equal and so the statistics for fractional collection with $\alpha = \beta = 0$ are left out of the analysis.

In every scenario, 522 planning problems are solved, one for each review period. Note that the possibility exists that no must-orders are triggered in a certain review period. In this case, no set partitioning problem is generated and solved. In addition, the number of must-orders is sometimes very low, resulting in very small set partitioning problems. Calculation is interrupted when the time exceeds ten minutes. If this happens, the best solution found in ten minutes is taken as the solution to the planning problems. These instances are not included in the average computing times mentioned in the tables.

Table 8.3: Set partitioning statistics with full collection: base scenario

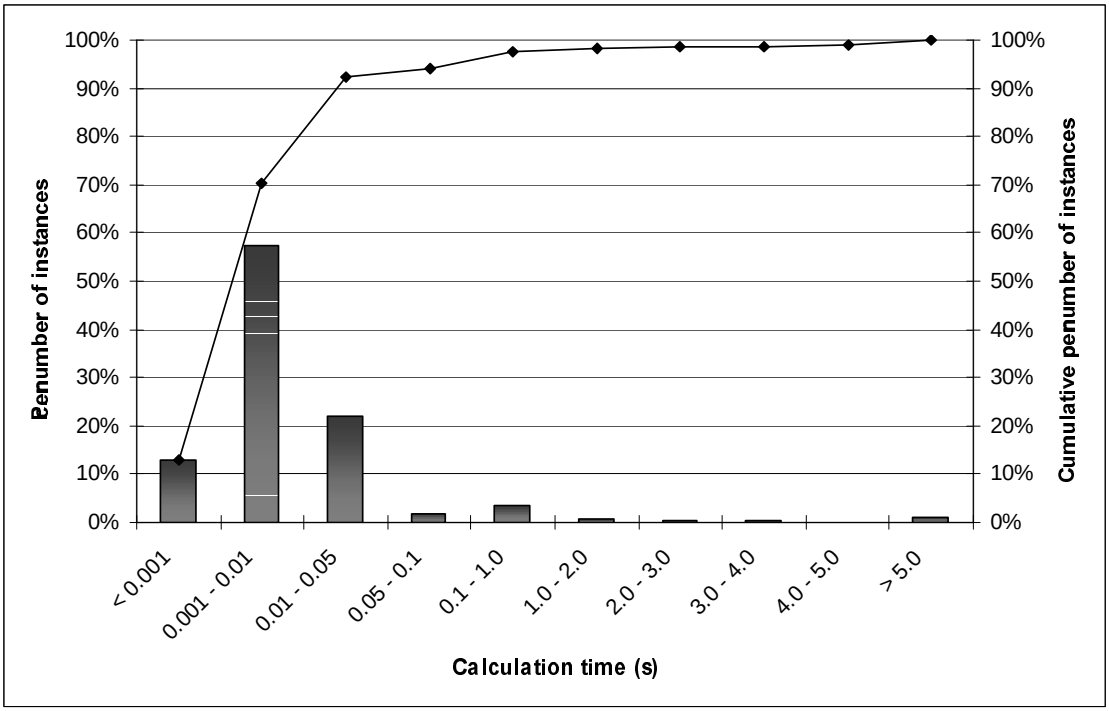
$\alpha = \beta$	0	1	2	3	4	5	6
Number of instances	522	521	520	522	518	515	518
Number of instances time > 10 min	3	0	0	0	0	0	1
Maximum number of rows	29	38	43	53	56	68	68
Maximum number of columns	10968	6522	6235	7591	7974	9833	27913
Average number of rows	9	14	20	25	31	36	41
Average number of columns	424	253	193	208	236	296	352
Average time (s)	3.61	2.66	0.48	1.45	0.34	0.99	0.85

Table 8.4: Set partitioning statistics with fractional collection: base scenario

$\alpha = \beta$	1	2	3	4	5	6
Number of instances	516	513	514	513	514	516
Number of instances time > 10 min	0	0	0	0	0	0
Maximum number of rows	35	46	51	58	66	72
Maximum number of columns	10443	16836	22391	28405	68178	38803
Average number of rows	14	20	25	31	36	42
Average number of columns	334	395	428	554	794	661
Average time (s)	1.73	1.74	0.08	0.11	0.48	0.39

The sizes of the problems when fractional collection is allowed are similar to the sizes in the situation where only full collection of can-orders is allowed. When the can-order level increases, two effects on the number of routes can be noted. First, with an increase in the number of can-orders, the number of must-orders triggered decreases, since orders are fulfilled proactively. With the decrease in the number of must-orders, the number of routes also decreases. On the other hand, with an increase in the number of can-orders, the number of possible combinations and the number of routes increases. These two effects account for the fluctuations in the number of columns when the size of α and β increases. Figure 8.7 shows the distribution of the computing times over all problems in both scenarios. The total number of set partitioning problems is 6,722, of which only 81, or 1.2%, take longer than 5 seconds to solve, while 6,329, or 94.2% take less than 0.10 seconds to solve.

Figure 8.7: Solution times of the set partitioning problems in the base scenario



8.5.2 Double truck capacity

When the capacity of a truck doubles, from 14 m^3 to 28 m^3, the capacity is less restrictive in the route generating process and we expect more routes to be generated. We therefore expect an increase in the size of the set partitioning problems, measured in number of columns. Tables 8.5 and 8.6 show the statistics of the set partitioning problems for these scenarios with full and fractional collection of can-orders, respectively.

Table 8.5: Set partitioning statistics with full collection: double truck capacity

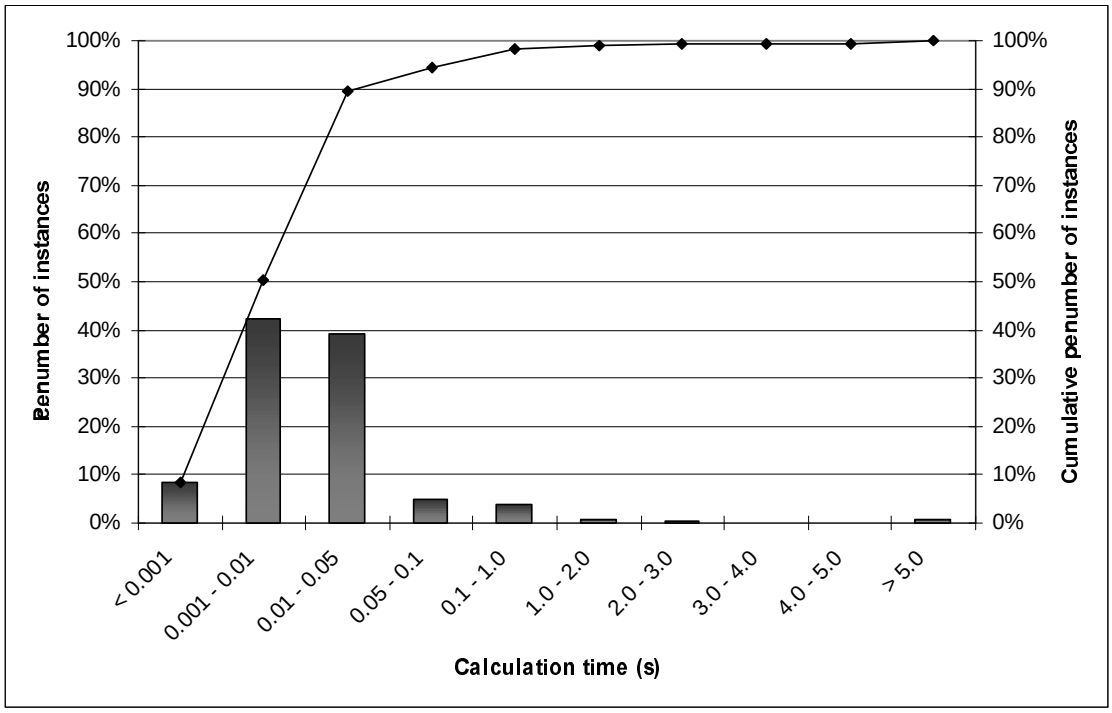
$\alpha = \beta$	0	1	2	3	4	5	6
Number of instances	522	516	513	500	504	504	492
Number of instances time > 10 min	1	0	0	1	0	0	0
Maximum number of rows	29	38	43	48	51	56	61
Maximum number of columns	39888	52429	40133	85147	145211	164934	165119
Average number of rows	9	13	17	23	27	32	37
Average number of columns	1422	965	760	1398	1340	1701	2421
Average time (s)	0.96	0.09	0.09	0.05	0.26	0.07	0.06

Table 8.6: Set partitioning statistics with fractional collection: double truck capacity

$\alpha = \beta$	1	2	3	4	5	6
Number of instances	520	509	506	500	502	492
Number of instances time > 10 min	0	0	0	0	0	0
Maximum number of rows	38	44	51	54	59	60
Maximum number of columns	43934	46054	161871	186744	186929	164634
Average number of rows	13	17	22	27	33	37
Average number of columns	906	732	1329	1869	2876	3237
Average time (s)	0.13	0.13	0.07	0.10	0.12	0.09

Indeed, we see that the average size of the set partitioning instances increases somewhat, while the computing times remain very small. On average, the solution times of these instances are even below the solution times of the instances in the base scenario. This can be explained by the observation that the variation in route costs is higher in this scenario than in the base scenario, making the set partitioning instances easier to solve. Figure 8.8 shows the distribution of the computing times over all problems in both scenarios. The total number of problems is 6580, of which only 36, or 0.5%, take longer than 5 seconds to solve, while 6210, or 94.4% of the problems take less than 0.10 seconds to solve.

Figure 8.8: Solution times of the set partitioning problems with double truck capacity



8.5.3 Review period of three weeks

When the review period is three times as long, meaning collection only takes place once every three weeks, we expect the number of must-orders per review period, and thus the size of the set partitioning instances, to increase. Tables 8.7 and 8.8 show the statistics of the set partitioning problems for these scenarios with full and fractional collection of can-orders, respectively.

Table 8.7: Set partitioning statistics with full collection of can-orders: review period of three weeks

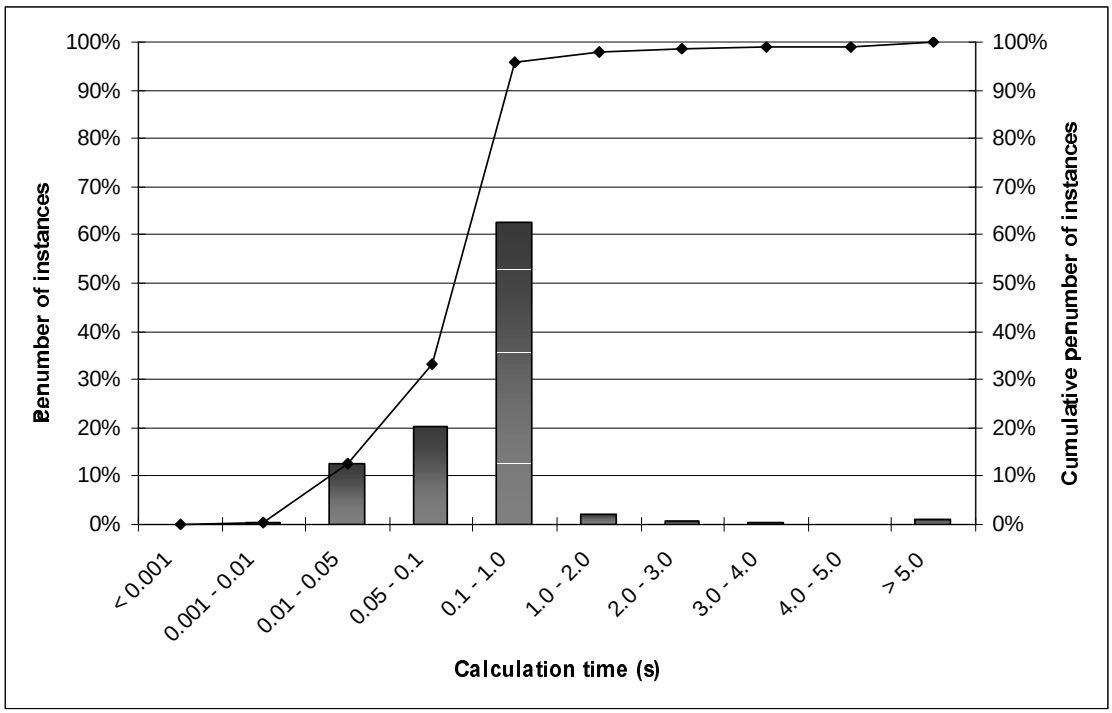
$\alpha = \beta$	0	1	2
Number of instances	174	174	174
Number of instances time > 10 min	7	1	0
Maximum number of rows	76	90	85
Maximum number of columns	526431	462762	135434
Average number of rows	27	46	65
Average number of columns	20065	7373	8500
Average time (s)	0.16	0.15	0.23

Table 8.8: Set partitioning statistics with fractional collection of can-orders: review period of three weeks

$\alpha = \beta$	1	2
Number of instances	174	174
Number of instances time > 10 min	1	0
Maximum number of rows	94	80
Maximum number of columns	2527870	158132
Average number of rows	46	65
Average number of columns	21981	19267
Average time (s)	0.19	0.51

As expected, the average size of the instances, as well as the computing times of the set partitioning problems, increases. Note that the maximum amount of time we look forward in the planning process is six weeks, which corresponds to $\alpha = \beta = 2$. Moreover, since the review period is longer, we have fewer planning instances and the maximal number of set partitioning problems that is generated during one scenario decreases from 522 to 174. Figure 8.9 shows the distribution of the computing times over all problems in both scenarios. The total number of problems is 870, of which 9, or 1.0%, take longer than 5 seconds to solve, while 833, or 95.7% of the problems take less than 1 second to solve.

Figure 8.9: Solution times of the set partitioning problems with a review period of three weeks



8.5.4 General statistics

In total, we have analyzed 14,172 problems in this section. The largest problem, measured in number of columns, has 94 rows and 2,527,870 columns. On average, the problems have 27 rows and 1878 columns.

From the total set, 15 problems, or 0.1%, were not solved within ten minutes. These instances have an average of 382,085 columns and 61 rows. For these instances we always find a solution, which is, on average, only 3.1% higher than the optimal solution. For five instances, the best solution found is equal to the optimal solution. With CPLEX 9.0, three out of the 15 instances are also not solved within ten minutes. The average solution time of CPLEX on the remaining 12 instances is equal to 98.1 seconds. Chapter 9 discusses a second case where LaRSS is used in a similar way and where the size of the problems is larger than the problems considered here. For this case we make a more detailed comparison between LaRSS and CPLEX.

An interesting observation is that all of the 15 problems that are not solved in ten minutes, have more must-orders than vehicle-day combinations, such that the index 'vd' in the problem [8.6] – [8.12] is of importance. As mentioned before, in most cases we can skip this index, since we know for sure that we will never have more routes than vehicle-day combinations. It seems that this extra index makes the problem relatively more difficult to solve. We will examine this with a small example.

Example

Suppose we have a planning problem with 15 must-orders and 20 can-orders and we have generated 2000 different routes to fulfill these orders. The number of vehicle-day combinations equals |VDC|. If this number were larger than or equal to 15, the number of must-orders, then the problem would reduce to:

$$\min \quad \sum_r c_r \cdot X_r \qquad [8.13]$$

Subject to

$$\sum_r a_{mo,r} \cdot X_r = 1 \qquad \forall mo \qquad [8.14]$$

$$\sum_r a_{co,r} \cdot X_r + sc_{co} = 1 \qquad \forall co \qquad [8.15]$$

$$X_r \in \{0,1\} \qquad \forall r \qquad [8.16]$$

$$sc_{co} \in \{0,1\} \qquad \forall co \qquad [8.17]$$

On the other hand, when |VDC| is smaller than 15, we would have the problem given by [8.6] – [8.12]. With ten vehicle-day combinations, the number of variables would then equal 20,045, while the problem with 15 vehicle-day combinations would have only 2,035 variables. However, the growth in the number of variables is not the most important complicating factor. Suppose that one optimal solution exists with five routes, while we have ten vehicle-day combinations at our disposal. These five routes can be driven on all vehicle-day combinations, such that the set partitioning problem

would have $\binom{10}{5}$= 252 optimal solutions. This number can become much larger in more advanced settings, which makes our branching strategy very slow, since all solutions have no columns in common and they all have the same costs.

From the 15 problems we examine, eight come from the scenario with review periods of 18 days. Since there are two vehicles available in these scenario's, we have 36 vehicle-day combinations. If, in one of these cases, we would have an optimal solution consisting of five routes, then the same solution would exist $\binom{36}{5}$ = 376,992 times in our set partitioning tableau. The seven remaining cases all have ten vehicle-day combinations.

If we have more general solvers at our disposal, we can easily rewrite the problem of [8.6] – [8.12] to a much simpler version:

$$\min \quad \sum_r c_r \cdot X_r \qquad [8.18]$$

Subject to

$$\sum_r a_{mo,r} \cdot X_r = 1 \qquad \forall mo \qquad [8.19]$$

$$\sum_r a_{co,r} \cdot X_r \leq 1 \qquad \forall co \qquad [8.20]$$

$$\sum_r X_r \leq |VDC| \qquad [8.21]$$

$$X_r \in \{0,1\} \qquad \forall r, vd \qquad [8.22]$$

This problem can be solved with the extended solver discussed in Section 10.2. Section 10.2.6 examines the performance of this extended solver on this type of problem. For the 15 instances under consideration, the performance of the extended solver is good: the solution times vary from 0.03 seconds to 19.74 seconds, with an average solution time of 2.02 seconds.

8.6 Concluding remarks

8.6.1 Business perspective

This chapter discussed an application of remote monitoring of inventory levels in reverse logistics. We examined the possibilities of Collector Managed Inventory (CMI), the reverse logistics counterpart of Vendor Managed Inventory (VMI). We developed a planning methodology to support the collection company in constructing operational planning schedules. Information coming from the telemetry units placed at the waste generator's site allowed us to foresee when collection should take place and to actively search for combination possibilities in planning collection trips, thereby

reducing the transportation costs.

The potential of collector managed inventory is illustrated by the results of the real-life project for the collection of oil and coolant at Auto Recycling Nederland. Cost savings amount to 18.9% when proactive collection planning is compared to the traditional reactive collection planning. The system is implemented in 2005, together with a similar system for the collection of fuel and windscreen washer fluid.

The attractiveness of remote monitoring and a proactive planning approach includes more than just the cost benefit illustrated in this case study. The new system also reduces the environmental burden caused by transportation emissions, as well as road congestion. In our case, the total number of kilometers driven can be reduced by 26.3% compared to the conventional reactive planning methodology.

8.6.2 Mathematical perspective

In the case discussed in this chapter, the operational planning problem is solved using LaRSS. For every scenario, a simulation run of ten years is performed, where a planning problem is solved in every review period, normally every week. Thus, for every simulation run, over 500 set partitioning problems are solved. This chapter evaluated the performance of LaRSS for different scenarios. LaRSS appears to work very well on the problems considered. In total, 14,172 problems are solved, with number of columns up to 2,527,870 and number of rows up to 94. Of the total set of instances, 15 problems were not solved within ten minutes. Excluding these instances, the average solution time of the remaining 14,157 problems is 0.6 seconds. These instances have an average of 1,475 columns and 27 rows. From the 15 problems not solved within ten minutes, 3 are also not solved in ten minutes by CPLEX. The average computing time of CPLEX on the remaining 12 instances is equal to 98 seconds. With the extended solver that will be discussed in Section 10.2, these problems can be solved to optimality in an average time of 2 seconds. With LaRSS, a solution is found within ten minutes for each of these 15 problems, which is on average 3% higher than the optimal solution. The average solution time over all 14,172 problems is then equal to1.3 seconds.

This project extends further than the case described in this chapter. In total, over 60 scenarios were calculated, considering not only oil and coolant, but also fuel and windscreen wiper fluids. In the latter case, the average size of the set partitioning problems is larger than in the case described here. In all cases, LaRSS was successfully applied to solve the operational planning problem.

Chapter 9

Case study: vehicle routing in the closed-loop container network of ARN

This chapter deals with a case study considering the collection and delivery of containers for Auto Recycling Nederland. This case study is performed at CentER Applied Research using LaRSS to solve set partitioning problems. The content of this chapter has appeared in revised form as Le Blanc et al. (2004B). For more general information about Auto Recycling Nederland (ARN), see Chapter 8 and Van Burik (1998).

The outline of this chapter is as follows. Section 9.1 discusses the problem setting. Section 9.2 reviews the literature on similar problems. In Section 9.3, the methodology and the corresponding heuristics are described. Section 9.4 covers the structure of the analysis. Section 9.5 discusses the results of the case study. Section 9.6 discusses the statistics considering the set partitioning problems solved. Finally, in Section 9.7, we draw some conclusions.

9.1 Introduction

9.1.1 Background

The case study deals with optimizing the collection of containers that are used to transport end-of-life materials from dismantled vehicles. Due to pressures from the market, the ARN system will need to improve further the reverse chain for the processing of end-of-life vehicles (ELVs). As chain director, ARN outsources the actual processes to existing ELV-dismantlers, shredder companies, recyclers and logistic service providers (LSPs). The LSPs, contracted for a period of three years, are responsible for the logistics activities in a certain province. Their activities include

the transportation of the containers to a depot, consolidation at the depot and transportation to the recycling company. Sometimes value-adding activities such as sorting are performed at the depot. The current logistic planning activities are decentralized and performed by the individually contracted LSPs. The LSPs are assigned to ELV-dismantlers on the basis of provincial boundaries. In a central planning scenario, transportation orders are not sent directly to individual LSPs, but are collected on a centralized level and assigned in clusters to the LSPs, making use of the cost benefits of combined orders. Allocation of ELV-dismantlers to LSPs is no longer fixed, but adjusted regularly based on the optimization of routes on a central level.

This study considers manually dismantled, high-volume materials stored and collected in containers. An ELV-dismantler that has a full container submits a request for collection to the logistic service provider (LSP). Within five working days, the LSP visits the dismantler and exchanges the full container for an empty one. Glass, rubber strips and PU-foam are collected in a compartmented container, specially designed for ARN. Tires and bumpers are collected in 35m3 containers for all ELV-dismantlers. Currently, all materials are brought to the depot. There, all materials, except tires, are sorted and processed and then transferred by bulk transport to recyclers, located for the most part in neighboring countries. Since tires need no processing at the depot and the four contracted recycling companies are located in the Netherlands, they can be sent directly to recyclers, bypassing the depot. Our computational experiments examine the cost benefits of this option. We focus on the planning of requests from ELV-dismantlers to have containers collected. Since the recyclers of materials other than tires are located abroad, transport of these materials to the recyclers usually takes the form of a linehaul trip. Linehaul trips offer no combination possibilities and the costs of these trips are assumed to be fixed.

Currently, LSPs use two types of lifting mechanisms for loading and unloading containers onto a truck. The first system uses an iron chain to drag the container up onto the truck, while the second system uses a pneumatic hook to pickup the container and place it on the truck. Although both systems work adequately, they are not compatible. A container or truck suitable for the hook system is not suitable for the chain system and vice versa. This restriction must be taken into account in planning the trips, since LSPs do not have both lifting mechanisms, which leads to a complexity-reducing separable structure. This study examines the cost benefits of standardizing this lifting mechanism.

9.1.2 Goal

The study aims to analyze and improve the system of collecting containers. We thus examine the following situations:

- Allowing direct shipment of containers from dismantler to recycler, bypassing the consolidation depot.

- Changing the allocation of dismantlers to LSPs from the current assignment, based on provincial boundaries, to optimal fixed assignment or to dynamic assignment based on optimal routing decisions in each planning period.
- Standardizing the lifting mechanism for loading and unloading containers onto a truck.

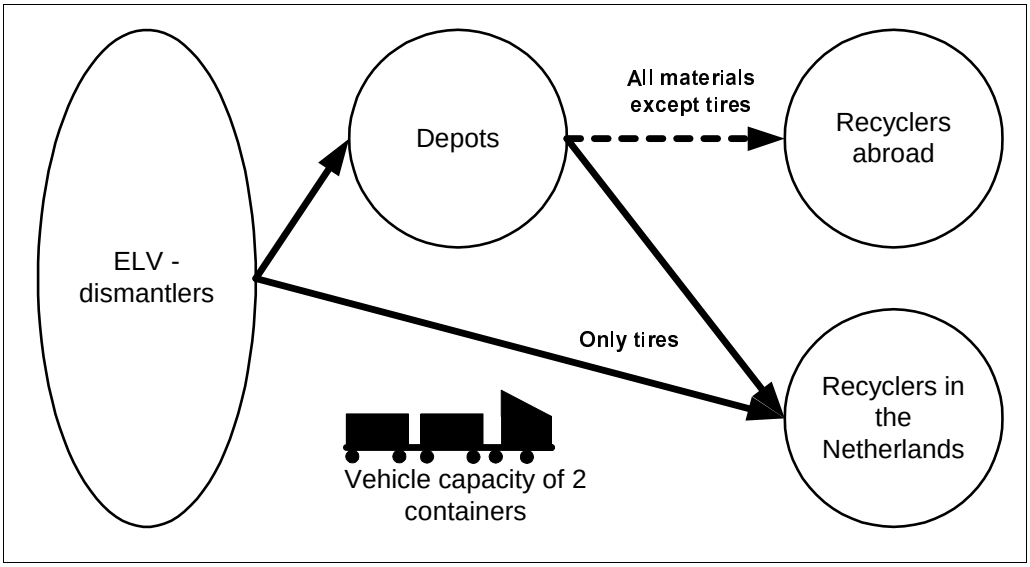
Although this is mainly a tactical study, we have chosen to solve the operational problem as well, to get a good estimate of transportation costs and performance, since the small nuances in different scenarios cannot be expressed adequately in tactical models. The problem resembles a unique multiple LSP vehicle routing model with pickup and delivery, allowing alternative delivery locations and with small vehicle capacity (two containers), which has not been described in the literature before. We call this the 2-container collection problem. The next section contains a formal description of the problem.

9.1.3 Problem formulation: the 2-container collection problem

The 2-container routing problem consists of a set of ELV-dismantlers, a set of depots, owned by an LSP and a set of recyclers. Distance and travel times between all locations are known. Both ELV-dismantlers and depots can initiate transportation orders for containers. At an ELV-dismantler, empty containers are exchanged for full ones, while at a recycling facility full containers are exchanged for empty ones of the same type. Since a shortage of containers never occurs in practice in a closed-loop system, the depot locations are assumed to have sufficient storage of all container types to exchange. Orders may be for either one or two containers; all orders concern containers of the same type. Full containers coming from ELV-dismantlers can be delivered either to a depot or to a recycling facility; full containers coming from a depot can be delivered only to a recycling facility. Which delivery location is selected depends on policy, practical restrictions, the estimated gate-fee for dropping the order at the location and the costs of including the delivery location in the route. The gate-fee depends on the residual value of the product and can even be negative, i.e. money is paid by the recycler to acquire the material. Figure 9.1 gives a conceptual mapping of the problem

A vehicle's route starts and ends at the depot. A route may take no longer than nine hours, one hour of which is overtime, at a 50% higher rate. Each stop involves a fixed stopping time and a variable loading and unloading time. The costs of a route are composed of a distance and a time component. The model allows for differentiating the kilometer and hourly rates per LSP. Vehicle capacity in the model is limited to two containers. Each LSP is deemed to have an unlimited number of vehicles. This is realistic since these types of trucks are widely used. The next section explores relevant literature dealing with similar problems.

Figure 9.1. Conceptual overview of the collection problem



9.2 Literature

The literature on vehicle routing is abundant; see Bodin et al. (1983) and Toth and Vigo (2002). In reverse supply chains, variants of the classical vehicle routing problem occur that have been less extensively studied (Dethloff 2001). Beullens (2001) provides an excellent overview of vehicle routing models and the special types of models occurring in reverse logistics.

The problem closest to the situation at hand is the skip problem (SP) as described in De Meulemeester et al. (1997). Vehicles start at a depot and must deliver empty skips to customers, collect full skips from customers and deliver the full skips to either the depot or one of the disposal facilities. A vehicle has the capacity to carry one skip at a time. Skips can be of multiple types and this is a restriction in exchanging full for empty. De Meulemeester et al. (1997) develop two heuristics and an exact procedure for solving this real-life problem. The exact procedure is based on enumeration. The first heuristic is based on the classical Clarke and Wright savings heuristic. The second heuristic calculates a solution to a formulated transportation problem, providing a lower bound to the optimal solution. The solution to the transportation problem is made feasible in a number of heuristic steps. On average, the variant of the Clarke and Wright savings algorithm performed best.

Bodin et al. (2000) describe a variant of the skip problem called the rollon-rolloff vehicle routing problem (RRVRP). In a RRVRP trip, a truck with a capacity for one container departs from a depot to serve customers who need a container placed, collected or exchanged (full for empty). The network consists of only one depot and one disposal facility and all containers are of the same type. Bodin et al. describe four heuristics for this problem. Their contribution is of a theoretical nature, since they only

test the heuristics using a set of randomly generated instances.

Archetti and Speranza (2004) describe another variant of the problem, the so-called 1-skip collection problem (1-SCP). As the name indicates, vehicle capacity is limited to one skip or container. Since Archetti and Speranza deal with a real-life problem, they consider several practical restrictions, such as multiple container types, time windows, different priorities for different customers and a limited fleet size. Archetti and Speranza develop a three-phase algorithm. In phase 1, the set of skips that needs to be collected that day is determined and ranked in priority. In phase 2, a solution for the subset of skips is constructed. In phase 3, the solution is further improved by using local search procedures.

Although some of the models come close to the situation at hand, none has the same characteristics. All of these models consider the vehicle capacity to be limited to precisely one skip or container instead of two as in our case. Extending the algorithms described in the literature to the situation with two containers is not trivial. Techniques known from more general vehicle routing models could be used; these techniques, however, do not exploit the discrete capacity of only two containers. In this chapter we thus develop a new heuristic for tackling the problem at hand.

9.3 Methodology

The heuristic we develop to handle the case is a two-step heuristic. In the first step, a large number of candidate routes is generated. In the second step, a combination of routes is selected, minimizing the costs of drawing up a complete route plan, while satisfying all the requirements. This combination of route generation and set partitioning is referred to in the vehicle routing literature as the set partitioning approach, see for example Fleuren (1988). This type of algorithm, where a promising set of possibilities is generated and a solution is found by set partitioning, is referred to as petal algorithms (Laporte et al. 2000). An alternative way of applying set partitioning in this setting is by using column generation, see, for example, Agarwal et al. (1989). Since we have a fast set partitioning solver at our disposal and our average number of orders per route is limited, we opt for an enumeration of a large set of feasible routes.

9.3.1 Route generation

The purpose of route generation is to construct a set of feasible routes, such that the route selection procedure can make a ‘good’ choice from the set. To tackle this multi-depot pickup and delivery problem with alternative delivery locations, we introduce the concept of root-orders and sub-orders. This is described in Section 9.3.1.1. While the number of feasible routes grows exponentially, we are satisfied with the generation of a promising subset of routes. To restrict the number of candidate routes

generated, we use the concept of order neighborhoods; this is the topic of Section 9.3.1.2. Finally, the route generation procedure is described in Section 9.3.1.3.

9.3.1.1 Root and sub-orders

To handle the pickup and delivery problem with alternative delivery locations and selection of logistic service providers, we distinguish root- and sub-orders. Every transportation order has a general root-order with location- and LSP-specific sub-orders. Since each sub-order has a unique pickup and delivery location, as well as a logistic service provider, our algorithm can proceed along the same lines as a standard pickup and delivery heuristic. Some constraints must be added, however, to ensure that only one sub-order is performed per root-order.

Example

ELV-dismantler WreckRec has a container of tires that needs to be transported either to the tire recycler TireRec or to a depot of a logistic service provider. There are two competing logistic service providers with a depot: LogOpt and LogCheap. This single root-order results in four sub-orders as shown in Table 9.1.

Table 9.1. The sub-orders in the example of WreckRec.

Sub-order	LSP performing the order	Pickup location	Delivery location
1	LogOpt	WreckRec	LogOpt depot
2	LogOpt	WreckRec	TireRec
3	LogCheap	WreckRec	LogCheap depot
4	LogCheap	WreckRec	TireRec

If a sub-order is selected with delivery to the depot, where delivery to the recycler was also an option, we have to correct the route costs for the future transportation costs from the depot to a recycler. In this situation, the sub-order generates a new root-order in the next planning period for the transport to the recycler. Since planning periods are short, three working days, this heuristic step is not a severe limitation. These costs are estimated using the equation [9.1].

$$CostCor_{so} = \alpha \cdot LHC_{so} \cdot Load_{so} \qquad [9.1]$$

where:

α = Correction factor between ½ and 1

LHC_{so} = Linehaul costs to deliver a container from the depot of sub-order so to the cheapest recycler in transportation costs and gate-fee.

$Load_{so}$ = Number of containers in sub-order so

The correction factor α expresses the combination possibilities for the transportation orders from depot to recycler. If $\alpha = 1$ no combinations are made and the full linehaul costs are charged to collect a single container. The perfect combination would be two containers from the depot to the recycler and two containers from an ELV-dismantler adjacent to the recycler back to the depot, which corresponds with $\alpha = ¼$. In our

implementation we use $\alpha = 0.8$, which follows from empirical analysis.

9.3.1.2 Neighborhoods

While the total number of feasible routes can be very large, up to several million, we use the concept of neighborhoods to limit the set of candidate routes. Every order has a set of neighbors, ordered on a distance-based criterion. When we add orders to a route, we consider only orders that are in the neighborhood of the route, which is the union of neighborhoods of the orders in the route. Formally, we describe this as follows. At the start of an empty route, every sub-order can be inserted. Since we develop a set of routes, each root-order can occur on several routes. For each sub-order we define a set of neighboring sub-orders belonging to different root-orders. Let nb_subord_{so} denote this set of neighboring sub-orders for sub-order so. $RouteSubOrders_r$ denotes the set of sub-orders in route r. The neighborhood of a route r, denoted as nb_route_r, is the union of the neighborhoods of the sub-orders in a route, i.e. $nb\_route_r = \bigcup_{so \in RouteSubOrd_r} nb\_subord_{so}$.

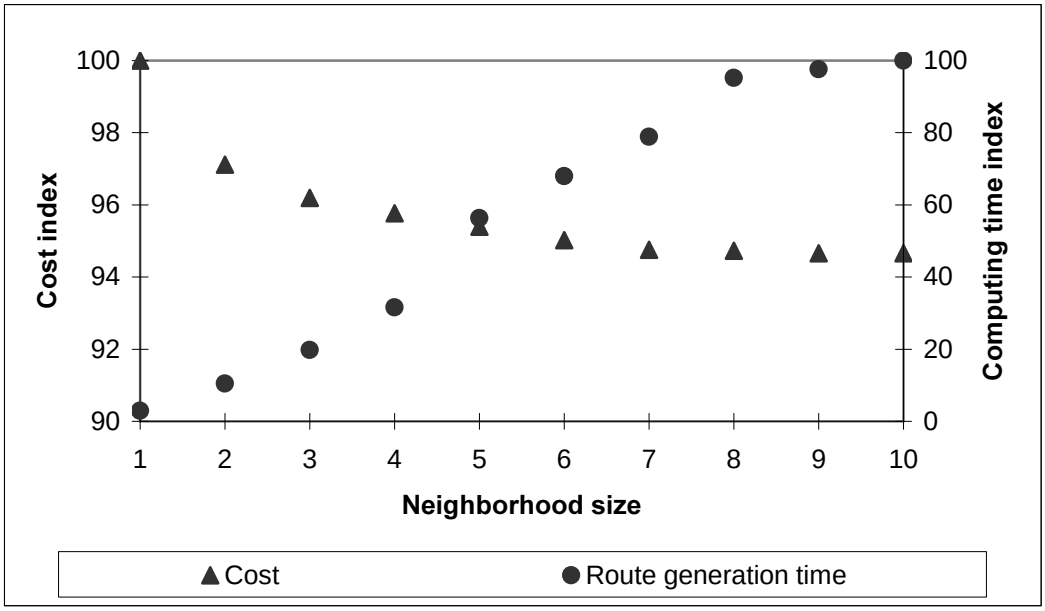
To determine the neighborhood of a sub-order, we need a distance measure. Consider two sub-orders so_A and so_B, with p_{so} and d_{so} denoting the respective pickup and delivery locations of sub-order so. Our distance measure is based on the best way to combine two orders rather than drive them separately. This criterion is expressed mathematically in [9.2].

$$\begin{aligned} dist_{so\_A,so\_B} = \min\{ & d(p_{so\_A}, d_{so\_A}) + d(d_{so\_A}, p_{so\_B}) + d(p_{so\_B}, d_{so\_B}), \\ & d(p_{so\_A}, p_{so\_B}) + d(p_{so\_B}, d_{so\_A}) + d(d_{so\_A}, d_{so\_B}), \\ & d(p_{so\_A}, p_{so\_B}) + d(p_{so\_B}, d_{so\_B}) + d(d_{so\_B}, d_{so\_A}), \\ & d(p_{so\_B}, d_{so\_B}) + d(d_{so\_B}, p_{so\_A}) + d(p_{so\_A}, d_{so\_A}), \\ & d(p_{so\_B}, p_{so\_A}) + d(p_{so\_A}, d_{so\_B}) + d(d_{so\_B}, d_{so\_A}), \\ & d(p_{so\_B}, p_{so\_A}) + d(p_{so\_A}, d_{so\_A}) + d(d_{so\_A}, d_{so\_B}) \} \\ & - d(p_{so\_A}, d_{so\_A}) - d(p_{so\_B}, d_{so\_B}) \end{aligned} \qquad [9.2]$$

For each sub-order, we list the distances to all sub-orders belonging to a different root-order and include the nearest nb_size sub-orders in nb_subord_{so}. Experiments with the required size of the neighborhood to find suitable solutions in acceptable computational time for the given study indicated that nb_size = 6 performs well; we use this value in the rest of this chapter.

Figure 9.2 shows the diminishing improvements found by extending the neighborhood size, using a representative sample of 25 real-life instances consisting of an average of 54 root-orders and 114 sub-orders. Further increasing the neighborhood size marginally improves the solution and causes a big increase in the route generation times. Note that above a certain threshold the route generation is no longer restricted and all feasible combinations are generated.

Figure 9.2. The influence of changing the size of the neighborhood on the quality of the solution, based on a representative sample of 25 real-life instances



9.3.1.3 Outline of the route generation algorithm

The route generator aims to create a large number of attractive and feasible routes. As stated in Section 9.3.1.2, we restrict the enumeration of routes by only appending orders from the neighborhood. A route is feasible if the maximum time allowed for one day and the maximum vehicle capacities along the route are not exceeded. Every time a full container is picked up from an ELV dismantler, it must be exchanged for an empty container of the same type. If this is not possible, the route is infeasible. We make use of a recursive function implementation for the systematic generation of routes. Figure 9.3 shows the outline of the route generation algorithm.

Figure 9.3: Outline of the route generation algorithm

```
Function RouteGenerator

IF ( Route empty )
        RouteNeighborHood := Set of all SubOrders
ENDIF
FOR ( SubOrder in RouteNeighborHood AND RootOrder unplanned ) DO
        InsertSubOrder( SubOrder )
        UpdateRouteNeighborhood
        IF ( RouteFeasible ) THEN
                WriteRouteToRouteSelectionProblem
                RouteGenerator
        ENDIF
        RemoveSubOrder
        UpdateRouteNeighborhood
ENDFOR
```

A sub-order is added to a route by inserting the pickup stop and the delivery stop of the sub-order in the route. For each possible position where the pickup stop (StopP) can be inserted, we find the cheapest position to insert the delivery stop (StopD), since we deal with the pickup and delivery situation. Figure 9.4 displays the main ideas behind the insertion of a sub-order in a route.

Although the number of routes generated is restricted by the size of the order neighborhood, it can still be very large in some cases. Occasionally, over 2.5 million routes are generated. In that case, because of memory limitations of our computers, we reduce the maximum allowed size of the neighborhood by one and restart the route generation.

Figure 9.4: Outline of the sub-order insertion function

```
Function InsertSuborder( SubOrder )

FOR ( Position in Route ) DO
        Insert StopP
        FOR ( Position in Route after Stop P ) DO
                Insert StopD
                UpdateRoute
                IF ( BestInsertion AND RouteFeasible ) THEN
                        StoreBestInsertionPosition
                ENDIF
                Remove StopD
        ENDFOR
        Remove StopP
ENDFOR
IF ( BestInsertionExists ) THEN
        Insert StopD and StopP at best position
        UpdateRoute
ENDIF
```

9.3.2 Route selection

The problem of finding the optimal combination of routes such that all orders are performed at minimal costs is formulated as a set partitioning problem. After some notation is introduced, the problem is given in equations [9.3] to [9.5].

Parameters

$\delta_{so,ro}$ = 1 if sub-order so belongs to root-order ro; 0 otherwise.

$a_{so,r}$ = 1 if sub-order so is contained in route r; 0 otherwise.

c_r = denotes the costs of driving route r in euros.

p_r = denotes the profit or costs (negative p_r) of route r as a result of the chosen delivery locations for the orders in route r in euros.

Variables

X_r = 1 if route r is selected; 0 otherwise.

The route selection problem

$$\min \quad \sum_r (c_r - p_r) \cdot X_r \qquad [9.3]$$

s.t.

$$\sum_r \sum_{so} (\delta_{so,ro} \cdot a_{so,r}) \cdot X_r = 1 \qquad \forall ro \qquad [9.4]$$

$$X_r \in \{0,1\} \qquad \forall r \qquad [9.5]$$

Note that $\sum_{so} \delta_{so,ro} \cdot a_{so,r}$ is either 0 or 1 by construction of the route generator and therefore the route selection problem is a pure set partitioning problem. We use LaRSS to solve the set partitioning problems. Section 9.6 examines the performance of LaRSS on these problems.

9.4 Structure of the analysis

9.4.1 Simulation

We use a simulation model to analyze the performance of the system. The transportation orders from ELV-dismantlers are generated following empirical distributions. To obtain representative results, each simulation run consists of ten replications of one year. In the simulation, the operational vehicle routing problem is solved twice a week for a planning horizon of three workdays. This means that over 1,000 set partitioning problems are generated and solved per simulation run.

Orders generated during a certain collection period are planned for and executed during the next planning period. For containers of tires brought to the depot, the orders for shipping the containers to the recycler are also issued at the beginning of the next planning period. Transportation orders are thus fixed at the beginning of a planning period.

9.4.2 Data and scenarios

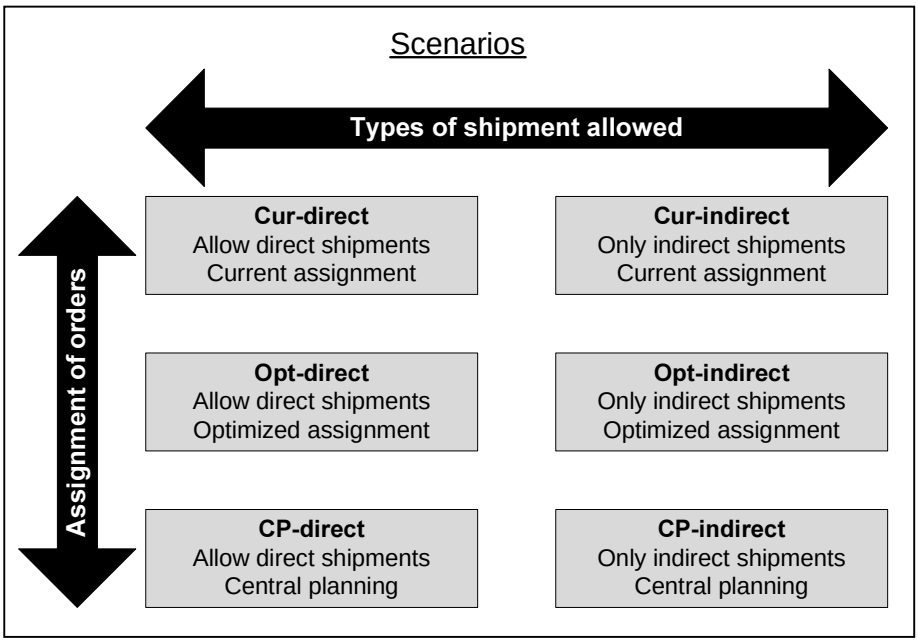
The scenarios are constructed in cooperation with the logistic experts of ARN and in cooperation with the logistic service providers hired by ARN. Distances and driving times used in the analysis were obtained from Evo-IT (Evo-IT, 2004). The cost figures used were obtained from the NEA (NEA, 2004), which is an authority on traffic and transportation issues in the Netherlands. We use cost prices rather than the commercial rates of individual LSPs. The data used for simulating the processes at

the ELV-dismantlers are empirical data available in the corporate databases of ARN. A detailed description of these data can be found in Schreurs (2004).
Scenarios are defined along three dimensions:

- The lifting mechanisms used by the LSPs:
    - The current situation: two different lifting mechanisms are used
    - The standardized situation: all LSPs use the same lifting mechanism
- The assignment of transportation orders to the logistics service providers
    - Current fixed assignment: ELV dismantlers are assigned to LSPs and recyclers on the basis of provincial boundaries.
    - Optimized fixed assignment: ELV-dismantlers are assigned to the closest LSP/recycler based on a distance criterion.
    - Central planning: no fixed assignment exists; the LSP with the best combination possibilities executes the transportation order.
- The allowed routes for containers of tires:
    - No direct shipment: all tire containers pass the depot.
    - Direct shipment: this is allowed if it is advantageous to ship tire containers directly to a tire recycler instead of the depot.

Figure 9.5 shows six scenarios defined along the last two dimensions and their scenario IDs. These six scenarios can be applied to both lifting mechanisms, the first dimension, resulting in a total of twelve. Scenario Cur-indirect is our reference scenario and corresponds to the current situation of ARN.

Figure 9.5: An overview of the scenarios



The current assignment of ELV-dismantlers to depots and recyclers is based on provincial boundaries, for historic reasons. In many cases, this assignment is far

from efficient, since provinces can have irregular shapes. We resolve this by simply assigning each ELV-dismantler to the nearest depot with the proper lifting mechanism. In the central planning scenario, the effect of a fixed assignment is analyzed by letting go of this restriction altogether and using dynamic planning on a centralized level.

Currently, nearly all tire containers are transported to the recycler via a depot, since the container must be weighed at the depot. Nowadays, recyclers also have accurate weighing facilities for trucks, making a stop at the depot redundant. Direct shipment of containers filled with tires is possible as long as the date of delivery is communicated.

9.5 Case results

9.5.1 Current logistic network

The results for the current logistic network with LSPs having different types of lifting mechanisms are presented in Table 9.2. For confidentiality reasons, the cost figures have been indexed. A comparison of the yearly indexed costs for the various scenarios is also presented in Figure 9.6.

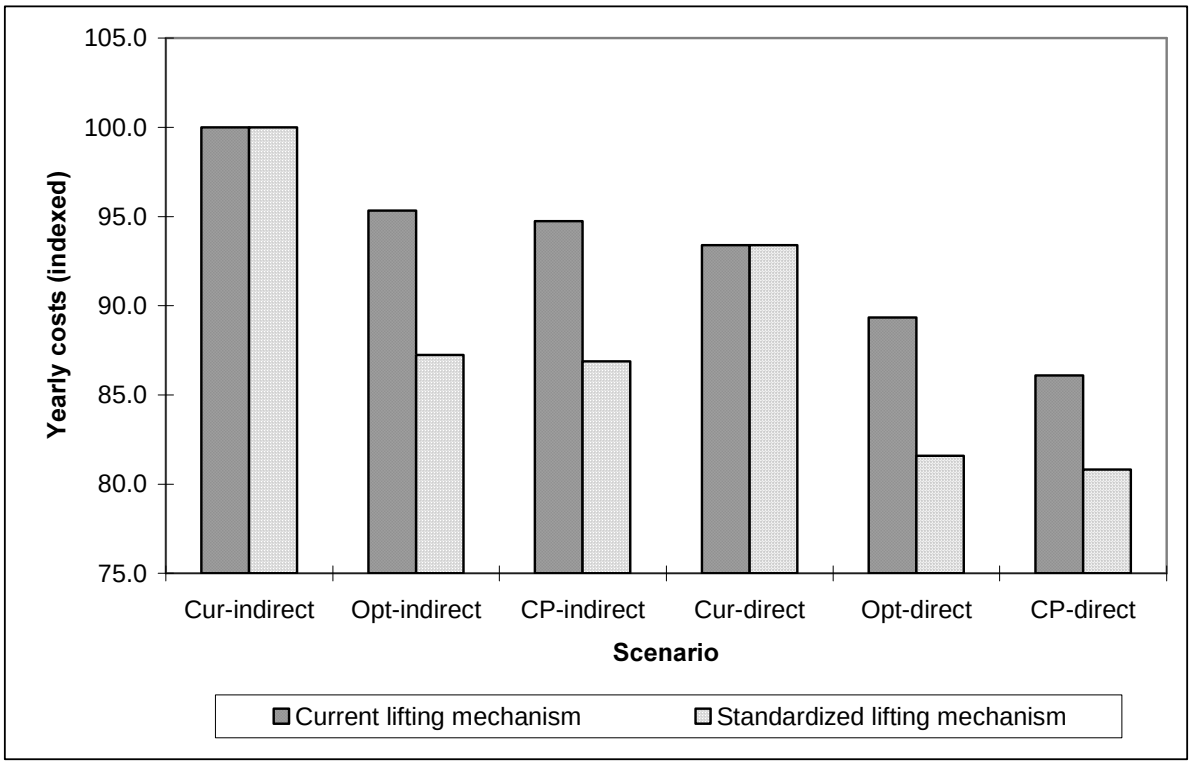
Table 9.2: Results for the current network with restrictions on the lifting mechanisms

Scenario ID	Cur-indirect	Opt-indirect	CP-indirect	Cur-direct	Opt-direct	CP-direct
Assignment	Fixed, current	Fixed, optimized	Free, central	Fixed, current	Fixed, optimized	Free, central
Type of shipments for tires	Only indirect	Only indirect	Only indirect	Allow direct	Allow direct	Allow direct
Avg. costs per year	100	95.3	94.8	93.4	89.3	86.1
Avg. distance per year (km)	505,779	471,610	467,188	483,092	458,972	433,735
Avg. # routes per year	2,887	2,906	2,907	2,346	2,336	2,226
Avg. # containers per route	2.45	2.44	2.44	2.39	2.32	2.42
Avg. route distance (km)	175.2	162.3	160.7	205.9	196.4	194.8
Avg. route duration (min)	291.0	277.6	276.1	331.3	319.7	325.4
Avg. driving time per route (min)	177.1	164.3	162.8	208.7	198.4	198.2
Avg. load-/unloadtime per route (min)	114.0	113.3	113.3	122.6	121.3	127.1

Allowing direct shipment of ship tire containers results in cost savings ranging from 6.3% to 9.1%, depending on the way in which ELV-dismantlers are assigned to LSPs. The average route length increases both in time and distance, since it is more attractive to make a small detour to drop tire containers at a tire recycler rather than bring them first to the depot and then to the recycler. This phenomenon causes the drastic decreases in the number of routes driven, since most tire containers are transported only once. Implementation of direct shipment is fairly easy and only requires some further arrangements with the recyclers.

Optimizing the assignment of ELV-dismantlers to depots and recyclers results in cost savings ranging from 4.4% to 4.7%. This effect is small, since the diversity in container lifting mechanisms allows little freedom for optimization. It is fairly easy to change to another fixed assignment: it merely requires renegotiation of contracts with LSPs. Compared to the optimal fixed assignment, the extra savings of dynamic allocation by central planning are limited, ranging from 0.6% to 3.6%. These marginal cost savings are not offset by the changes in the planning and control mechanisms to implement dynamic assignment.

Figure 9.6: Comparison of scenarios with current and standardized lifting mechanism



9.5.2 Network with uniform lifting mechanism for containers

The differences in lifting mechanisms used by logistic service providers are likely to cause inefficiencies. ARN is lobbying for standardizing container lifting mechanisms at the LSPs. This situation is compared to the current situation in Table 9.3. Figure 9.5 shows the yearly indexed costs of the various scenarios for the current situation as well as for uniform lifting mechanisms. Currently, the assignment of dismantlers to depots and recyclers takes into account differences in lifting mechanisms. Standardization of the lifting mechanism only makes sense, therefore, when the assignment is changed. We compare the current situation with the optimized assignment and central planning scenarios with a uniform lifting mechanism.

Table 9.3: Results for the current network after loosening the restrictions on the lifting mechanism

Scenario ID	Cur-indirect	Opt-indirect	CP-indirect	Cur-direct	Opt-direct	CP-direct
Assignment	Fixed, current	Fixed, optimized	Free, central	Fixed, current	Fixed, optimized	Free, central
Type of shipments for tires	Only indirect	Only indirect	Only indirect	Allow direct	Allow direct	Allow direct
Avg. costs per year	100	87.2	86.9	93.4	81.6	80.8
Avg. distance per year (km)	505,779	411,893	408,954	483,092	402,125	394,886
Avg. # routes per year	2,887	2,891	2,876	2,346	2,254	2,280
Avg. # containers per route	2.45	2.45	2.47	2.39	2.39	2.36
Avg. route distance (km)	175.2	142.5	142.2	205.9	178.4	173.2
Avg. route duration (min)	291.0	258.8	259.4	331.3	306.6	301.1
Avg. driving time per route (min)	177.1	145.1	145.0	208.7	181.4	177.2
Avg. load-/unloadtime per route (min)	114.0	113.7	114.4	122.6	125.1	123.9

Using optimal fixed assignment, the cost savings of standardizing the lifting mechanism are about 8.7%, when we allow direct shipments. If direct shipments are not allowed, the cost savings are 8.5%.

The cost savings of standardizing the lifting mechanism in the case of central dynamic planning are 8.3% when direct shipment is not allowed and 6.1% when direct shipment is allowed. Given standardized lifting mechanisms, the cost savings of dynamic central planning over optimized fixed assignment are less than 1%, regardless of whether direct shipment is allowed or not, which does not offset the costs of the organizational changes. Standardizing the lifting mechanism is comparable with increasing the network density for the LSPs. Improving the combination possibilities in a dense network has a marginal effect on the costs since, in a dense network, there are already abundant combination possibilities.

When we optimize the assignment of recyclers to LSPs, standardization of the lifting mechanism results in considerable cost savings that justify the necessary investment to implement this in the ARN chain.

9.6 Set partitioning results

This section discusses some statistics and results for the set partitioning problems solved for this case study. We will examine separately all twelve scenarios that are given in Tables 9.2 and 9.3. For every scenario, ten years are simulated with 106 review periods, resulting in a total of 12,720 set partitioning problems. For every set partitioning problem, calculation is stopped when the computing time exceeds five minutes. These experiments are performed on a normal desktop computer running on MS Windows 2000 with a dual 3,000 MHz Pentium processor and 3,072 MB RAM.

Table 9.4 shows the results of LaRSS on the scenarios with restrictions on the lifting mechanism. These scenarios correspond to the case results given in Table 9.2. Especially the factor of allowing direct shipments of tires appears to have a large

impact on the performance of LaRSS. When we allow direct shipments, we see three effects on the set partitioning problems:

1. The number of orders, or rows of the set partitioning tableau, decreases. This is caused by the fact that an indirect shipment of a tire container results in two orders in the long run: one transported from the dismantler to the depot and one from the depot to the recycler.
2. The number of routes, or columns of the set partitioning tableau, increases. This is caused by the fact that more possibilities exist when both direct and indirect shipments of tire containers are allowed.
3. The average computing time of the set partitioning problems decreases drastically. This is caused by the fact that direct shipment of tire containers is much cheaper than indirect shipment, resulting in clearer cost differences between columns and a more "obvious" optimum.

Another observation arising from these results is that the set partitioning problems belonging to the current allocation of dismantlers to LSPs are easier to solve than the problems belonging to the optimal assignment and central planning scenarios. This is caused by the fact that the current assignment is based on provincial boundaries, which are quite irregularly shaped, resulting in fewer possible routes.

Although the average number of columns is the largest in the last scenario, the performance of LaRSS is best on these instances. There is no obvious correlation between the number of rows or columns and the computing times of LaRSS. The largest problem solved for these scenarios contains 947,007 columns and 68 rows. This problem is solved in 4 seconds.

Table 9.4: Set partitioning statistics on the scenarios with restrictions on the lifting mechanism

Scenario ID	Cur-indirect	Opt-indirect	CP-indirect	Cur-direct	Opt-direct	CP-direct
Assignment	Fixed, current	Fixed, optimal	Free, central	Fixed, current	Fixed, optimal	Free, central
Type of shipments for tires	Only indirect	Only indirect	Only indirect	Allow direct	Allow direct	Allow direct
Number of instances	1060	1060	1060	1060	1060	1060
Number of instances time > 5 min	65	138	151	5	16	0
Average time remaining instances (s)	10.796	19.859	21.102	2.011	2.804	1.853
Average number of rows	62	62	62	53	52	52
Average number of columns	4823	5997	26953	38937	52353	293655
Maximum number of rows	90	88	87	76	76	86
Maximum number of columns	21164	24006	159963	666663	947007	810734
Average density of rows	230.13	294.28	1430.90	3195.09	4396.96	22658.10
Average density of columns	2.89	3.00	3.20	3.69	3.84	3.86

Table 9.5 shows the results of LaRSS on the scenarios without restrictions on the lifting mechanism. These scenarios correspond to the case results given in Table 9.3. These results give the same impression as Table 9.4. The largest problem solved in these scenarios contains 1,484,578 columns and 62 rows and is solved in 5

seconds.

Table 9.5: Set partitioning statistics on the scenarios without restrictions on the lifting mechanism

Scenario ID	Cur-indirect	Opt-indirect	CP-indirect	Cur-direct	Opt-direct	CP-direct
Assignment	Fixed, current	Fixed, optimal	Free, central	Fixed, current	Fixed, optimal	Free, central
Type of shipments for tires	Only indirect	Only indirect	Only indirect	Allow direct	Allow direct	Allow direct
Number of instances	1060	1060	1060	1060	1060	1060
Number of instances time > 5 min	48	102	96	5	8	1
Average time remaining instances (s)	11.284	20.166	20.169	1.835	5.083	2.903
Average number of rows	62	62	62	53	52	52
Average number of columns	4111	6378	66535	40730	82610	424259
Maximum number of rows	90	87	87	75	75	80
Maximum number of columns	14047	34264	551038	1484578	1155444	1205606
Average density of rows	193.33	318.24	3623.74	3389.42	6769.44	31260.63
Average density of columns	2.88	3.07	3.33	3.73	4.00	3.72

For the 6,360 set partitioning instances of the base scenario with restrictions on the lifting mechanism, we compared the performance of LaRSS and CPLEX. Table 9.6 shows the main results of this comparison. Note that the average time is always calculated over the instances that are solved to optimality within five minutes by both solvers.

Table 9.6: Comparison between the performance of LaRSS and CPLEX in the base scenario

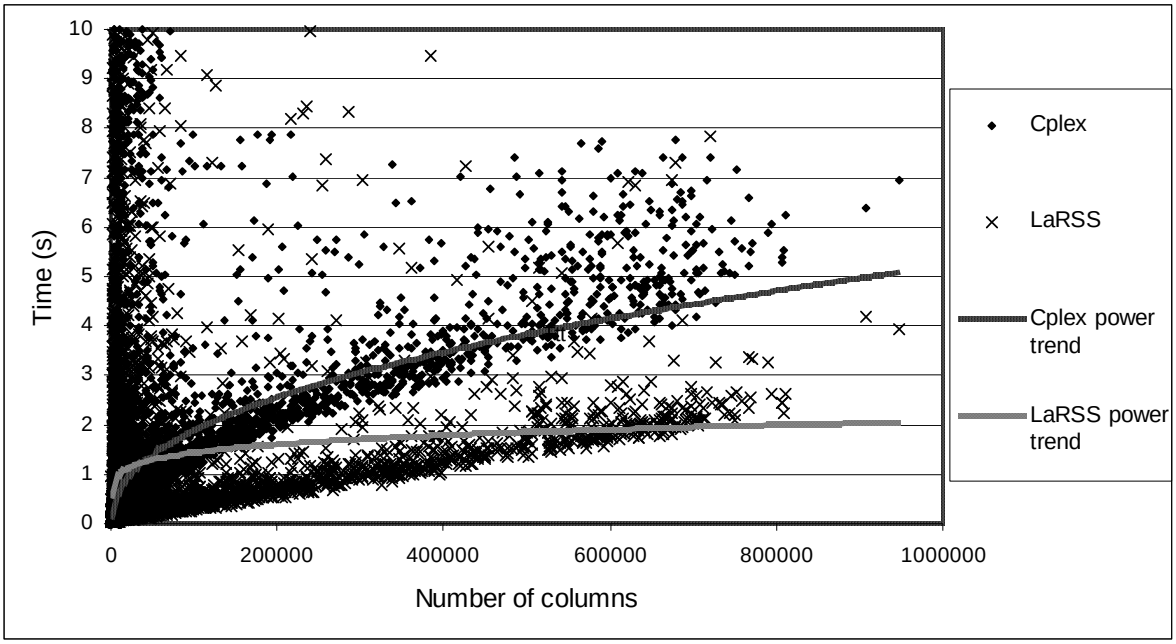
	CPLEX	LaRSS
Number of instances	6360	6360
Number of instances > 5 min	92	375
Number of instances solved to optimality by both solvers	5974	5974
Average time	4.32	8.97
Number of instances > 5 min Cur-indirect	8	65
Average time Cur-indirect	1.67	10.80
Number of instances > 5 min Opt-indirect	38	138
Average time Opt-indirect	3.03	19.55
Number of instances > 5 min CP-indirect	30	151
Average time CP-indirect	3.34	20.49
Number of instances > 5 min Cur-direct	7	5
Average time Cur-direct	3.49	1.70
Number of instances > 5 min Opt-direct	9	16
Average time Opt-direct	4.01	2.38
Number of instances > 5 min CP-direct	0	0
Average time CP-direct	9.24	1.14

In the overall performance, CPLEX performs better on average and is more robust,

meaning that there are fewer outliers in computing times. In 92 of the 6,360 cases, CPLEX is not finished within five minutes, while this number is 375 for LaRSS. The average time on the 5,974 instances solved to optimality by both solvers is equal to 4.3 seconds for CPLEX and 9.0 seconds for LaRSS.

In the three scenarios in which direct shipment of tire containers is not allowed, CPLEX performs much better than LaRSS, while for the direct shipment scenarios, LaRSS performs slightly better. Especially in the last scenario, where both solvers solve all 1,060 set partitioning problems to optimality within five minutes, LaRSS is much faster on average. Since the problem sizes, measured in the number of columns, are largest in these instances, the performance of LaRSS seems to be less sensitive to these sizes. Figure 9.7 shows the computing times in seconds against the number of columns for both LaRSS and CPLEX, for all problems that are solved within ten seconds, which is over 80% of the total number of problems. To illustrate the behavior of the two solvers, we have estimated a power function to describe the trend of the computing time. Compared to exponential, linear and polynomial functions, the power function; $y = ax^b$, appears to give the best fit to these data. As indicated, LaRRS performs worse when the number of columns is small, while the performance relative to CPLEX improves when the number of columns increases. Note that these observations are only based on the problem set considered here and can not be generalized to the whole class of set partitioning problems. In general, we can say that the performance of LaRSS depends more on the structure of the problem and the cost structure of the routes than on the problem size. The performance of CPLEX is more robust with regard to structure of routes and costs.

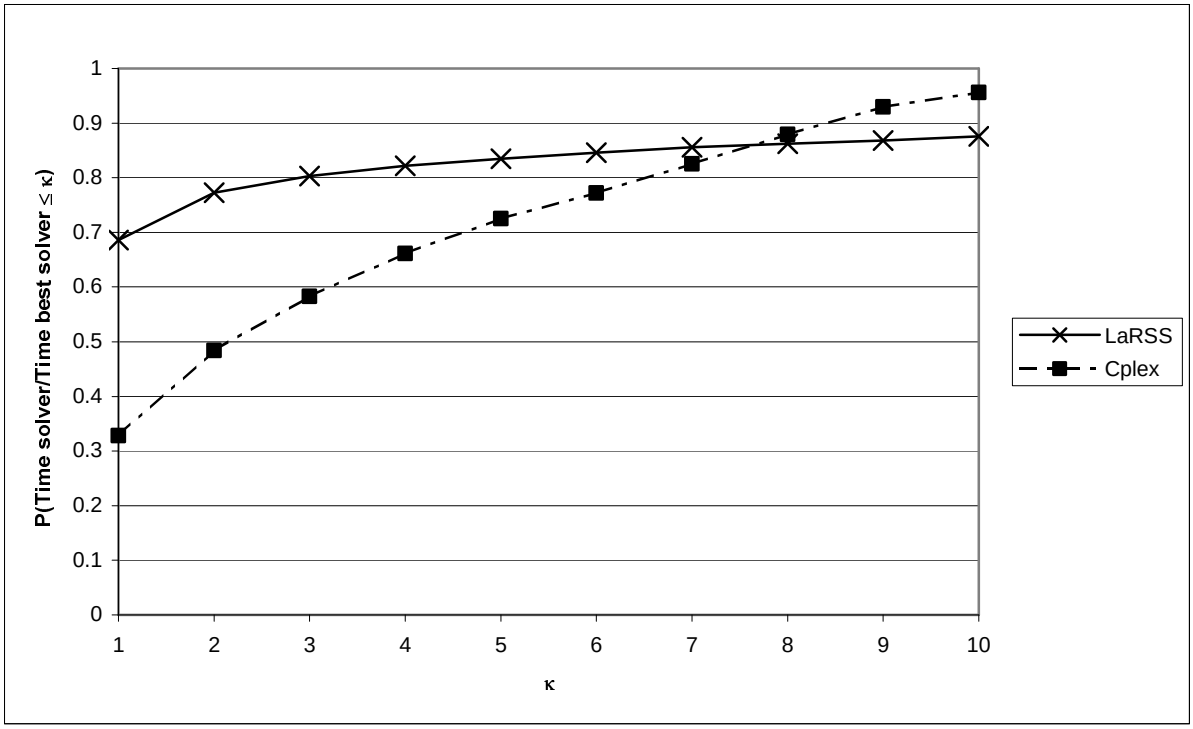
Figure 9.7: Computing times of CPLEX and LaRSS



From the 12,720 set partitioning problems, 635 are not solved to optimality by LaRSS. For all of these instances, at least one integer solution is found and the best solution found is taken as solution to the routing problem. In the base scenarios, 375 problems are not solved to optimality within five minutes by LaRSS, 81 of these are also not solved within five minutes by CPLEX. On average, CPLEX solves the remaining 294 instances within 40 seconds. For 138 or 47% of these problems, the solution found by LaRSS is equal to the optimal solution. On average, the solution found by LaRSS for these problems is only 0.14% away from the optimal solution.

Figure 9.8 shows a performance profile of LaRSS and CPLEX on this set of problems. As discussed in Chapter 7, a performance profile indicates for both solvers the probability, based on the current test set, that the computing time of the solver is within κ times the time of the best solver (Dolan and Moré, 2002). The picture indicates that LaRSS performs best on 69% of the problems, while CPLEX performs best on 33% of the problems. For 80% of the problems, the computing time of LaRSS is within a factor 3 of the best, while for CPLEX, this is the case for 58% of the problems. We can conclude that LaRSS performs quite well compared to CPLEX on this set of problems. However, the crossing of the two lines at $\kappa = 8$ indicates that LaRRS has more problems with a relatively high calculating time than CPLEX and thus that CPLEX performs more robustly on this set of problems.

Figure 9.8: Performance profile of LaRSS and CPLEX on the case test set



9.7 Concluding remarks

9.7.1 Business perspective

In this chapter we have described a real-life project in optimizing the logistic network for containers with materials from end-of-life vehicles. The underlying vehicle routing model is a unique multi-depot pickup and delivery model with alternative delivery locations. The heuristic we used is based on generation of a set of promising routes and selection of the optimal combination of routes by solving a set partitioning problem.

We analyzed the consequences, from a business point of view, of a better assignment of waste generators to logistics service providers (LSPs) and of routing decisions made by central planning. Furthermore, we analyzed the influence of a policy that did not allow the direct shipment from waste generator sites to recycling facilities and the effects of the different lifting mechanisms used for containers.

With respect to the assignment of recyclers to LSPs, we recommend changing the current fixed assignment, which is based on provincial boundaries, to the optimal fixed assignment. Considerable effort would be involved in implementation of the dynamic assignment option, while the additional savings over the optimal fixed assignment would be limited. Since the study shows that allowing direct shipment will result in cost savings and the organizational burden is not very large, we recommend allowing direct shipment of tires to recyclers. With respect to the lifting mechanism, the study showed that standardization would result in significant cost savings, making it worthwhile to standardize the lifting mechanism in the ARN network. Compared to the current system, the recommended new system, with standardized lifting mechanism, the option of direct shipments and the optimal fixed assignment, results in total cost savings of over 18%.

9.7.2 Mathematical perspective

In the case discussed in this chapter, the operational planning problem is solved using LaRSS. For every scenario, a simulation run of ten years is performed, where a planning problem is solved for every review period. This means that over 1,000 set partitioning problems are solved for every simulation run. Section 9.6 examined the performance of LaRSS on twelve different scenarios. LaRSS appears to work quite well on the problems considered. In total, 12,720 problems are solved, with number of columns up to 1,484,578 and number of rows up to 90. Of the total set of instances, 635 problems are not solved within five minutes. For these instances, the best solution found in five minutes is taken as the solution to the planning problem. In all cases at least one solution is found in five minutes. For the 294 instances where

the optimal solution is known, the best solution found is, on average, only 0.14% away from the optimum.

The average solution time of LaRSS on the remaining 12,085 problems is 9.24 seconds. The problems have an average of 57 rows and 89,820 columns. The scenarios that allow for direct shipment of tire containers are solved much faster than those that do not allow direct shipment, although the set partitioning problems belonging to the latter are much smaller on average. The most important influence on the solution times of the set partitioning problems seems to be the cost structure of the columns; when the variance in costs of the different columns, or routes in this case, is small, the set partitioning problems are more difficult to solve.

For the six scenarios with restrictions on the lifting mechanism, a comparison is made between the performance of LaRSS and CPLEX. On average, CPLEX performs better and more robust. When the number of columns grow larger, however, LaRSS performs increasingly better than CPLEX. This is probably caused by the extensive use of preprocessing in LaRSS, which can be crucial in reducing the solution time of large set partitioning problems.

Chapter 10

Extensions

This chapter deals with two extensions of LaRSS. We first examine how LaRSS can be adjusted to solve problems that have a mix of set partitioning and set packing constraints. Second, we examine set partitioning problems with more general side-constraints.

10.1 Set packing constraints

This section examines the set of problems that have a mix of set packing and partitioning constraints:

Min $\sum_{j \in J} c_j \cdot x_j$ [10.1]

Subject to:

$\sum_{j \in J} a_{rj} \cdot x_j = 1 \qquad \forall r \in R^{SPP}$ [10.2]

$\sum_{j \in J} a_{rj} \cdot x_j \leq 1 \qquad \forall r \in R^{SPC}$ [10.3]

$x_j \in \{0,1\} \qquad \forall j \in J$ [10.4]

Here, $R = R^{SPP} \cup R^{SPC}$ is the set of rows (constraints) of the problem and J is the set of columns (variables). R, J and $\{a_{rj}\}$ are defined analogously to the set partitioning problem as discussed in Section 1.1. Sections 10.1.1 to 10.1.5 review all the techniques used to solve set partitioning problems that are discussed in this thesis and examines how these techniques have to be adjusted to solve the more general problems considered here. Note that the extra set packing constraints given in [10.3] can also be dealt with by introducing slack variables into the constraints to make the problem a pure set partitioning problem. Section 10.1.6 considers computational experiments of the extended solver and compares them to the results of this set

partitioning approach.

10.1.1 Preprocessing

We examine the preprocessing techniques used in LaRSS one by one. The adjustments needed for the four pure preprocessing techniques, equal columns, equal rows, contained rows and clique, are very small and easily incorporated. The row combination technique, however, is not applicable to inequality constraints.

Equal columns
The equal columns rule can be used without adjustments.

Equal rows
If two rows are covered by the same set of columns and both rows have the same sign, then one of these rows can be deleted. If two rows with different signs are covered by the same set of columns, then the row with the inequality sign can be removed from the problem. More formally:
If $J(r) = J(s)$ for $r,s \in R^{SPP}$ or $r \in R,\ s \in R^{SPC}$, then row s can be removed from the problem.

Contained rows
If row r is contained in row s, then we can delete all columns that cover s, but not r, only if row r is an equality constraint. In this case we can delete row s, regardless of the sign of this row. More formally:
If $J(r) \subseteq J(s)$ for $s \in R,\ r \in R^{SPP}$, then all $j \in J(s)/J(r)$ and s can be removed from the problem.

Clique
If all columns that cover row r, being an equality constraint, have one or more elements in common with a column j that does not cover row r, then we can remove column j, since choosing this column in a solution set will leave constraint r unsatisfiable.

Row combination technique
The concept of the row combination technique is based on the fact that when we combine two rows, we can delete all columns that cover only one of those two rows. This is no longer the case when one of the rows has an inequality sign and combining the rows makes no sense. The row combination technique is therefore only useful for pairs of equality constraints.

10.1.2 Lagrangian relaxation and dual heuristics

If we are dealing with a pure set partitioning problem, there is no restriction on the Lagrangian multipliers λ_r in the Lagrangian relaxation problem. However, when we are dealing with inequality constraints, the Lagrangian multipliers are restricted to be non-positive. The Lagrangian relaxation problem is now given by:

$$z_{LR}(\lambda) = \min \sum_{j \in J} \left(c_j - \sum_{r \in R} a_{rj} \lambda_r \right) \cdot x_j + \sum_{r \in R} \lambda_r \qquad [10.5]$$

Subject to:

$$x_j \in \{0,1\} \qquad \forall j \in J \qquad [10.6]$$

$$\lambda_r \text{ unrestricted} \qquad \forall r \in R^{SPP} \qquad [10.7]$$

$$\lambda_r \leq 0 \qquad \forall r \in R^{SPC} \qquad [10.8]$$

In the dynamic convergent subgradient search algorithm, we fulfill this extra restriction by using the following update function:

$$\lambda_r^{k+1} = \min\left(\lambda_r^k + \text{stepsize}^k \cdot g_r^k, 0\right) \qquad [10.9]$$

instead of the update function [3.29] as proposed in Chapter 3:

$$\lambda_r^{k+1} = \lambda_r^k + \text{stepsize}^k \cdot g_r^k \qquad [10.10]$$

Here, the gradient g^k and stepsize^k are the same as defined in Section 3.2.3. This adjustment is done only for rows r in R^{SPC}.

Dual improvement heuristic

In the dual improvement heuristic, the extra requirement on the Lagrangian multipliers can be applied straightforwardly. If for a certain row r in R^{SPC} it is true that all columns that cover r have strictly positive reduced costs, we can raise u_r with

$$\Delta = \min\left(-u_r \ , \ \min_{j \in J(r)} cr_j \right) \qquad [10.11]$$

In this way we ensure that u_r remains non-positive for all rows r. For all rows in R^{SPP} no adjustment is necessary.

Dual 3OPT heuristic

The dual 3OPT heuristic can be applied in the way described in Section 3.4.2, with the only requirement that if we have found three rows r_1, r_2 and r_3 for which the conditions [3.39] and [3.40] hold, we determine the maximum allowed value of Δ such that the constraints in [3.5] hold for all columns and the resulting vector λ remains dual feasible and non-positive for all r in R^{SPC}.

10.1.3 Primal heuristic

The primal heuristic discussed in Chapter 4 is a greedy heuristic, based on adding

columns to a partial solution until a feasible solution is found. This concept can be applied to the problems considered here as well. In the adjusted version, we use the same approach as depicted in Figure 4.1 and the same row orderings as introduced in Section 4.1.2. The only adjustment we make is that in every iteration we do not take just the next row in the row ordering, but we take the next row that has an equality sign.

10.1.4 Branch and bound

We take the dynamic constraint-based branching method of Section 5.3.2 and adjust it for the problems considered here. Similar to the primal heuristic, the way in which we choose the next row to be covered in a certain iteration needs to be adjusted. As long as there are set partitioning constraints that are not covered, we use the unadjusted dynamic constraint-based branching method. When all constraints in R^{SPP} are covered, we register the solution obtained at that point and start branching the rows in R^{SPC} in the same dynamic way, to ascertain whether it is favorable to cover these rows. This can be the case since we allow the costs of a column to be negative.

10.1.5 Other adjustments

The most complicated aspect of the mixed set partitioning/set packing problems does not lie in one of the techniques discussed above, but in the use of the reduced costs throughout the solver. In the standard set partitioning problem, given a partial solution $x \in \{0,1\}^{|J|}$, the induced subproblem with row set R' and column set J' and a dual feasible vector u^f, a lower bound on the optimal solution is given by:

$$\sum_{j \in J} cr_j \cdot x_j + \sum_{r \in R} u_r^f \qquad [10.12]$$

When x is a feasible solution, [10.12] gives the value of the solution. Now suppose we have a mixed set packing/partitioning problem with row set $R = R^{SPP} \cup R^{SPC}$, column set J, a partial solution $x \in \{0,1\}^{|J|}$ and induced subproblem with row set $R_{IS} = R_{IS}^{SPP} \cup R_{IS}^{SPC}$ and column set J'. Here, R_{IS}^{SPP} contains the set partitioning constraints that are not covered and R_{IS}^{SPC} are the set packing constraints that are not covered. For every partial solution $x \in \{0,1\}^{|J|}$, the value given by [10.12] provides a lower bound to the problem. However, when x is a feasible solution to the mixed set partitioning/packing problem, this value does not equal the value of the solution and has to be corrected for the set packing constraints that are not equal to 1:

$$z^{solution} = \sum_{j \in J} cr_j \cdot x_j + \sum_{r \in R} u_r^f - \sum_{r \in R_{IS}^{SPC}} u_r^f \qquad [10.13]$$

This complication influences particularly the performance of the branch and bound

procedure, since this algorithm depends heavily on the use of the reduced costs. To solve this problem, we correct the solutions we find in the branch and bound routine for the non-covered set packing rows. This has an obvious disadvantage: since the value of the solution is underestimated during the branch and bound routine, we have to go much “deeper” in the branch and bound tree to discover whether the solution is indeed better than the upper bound known at that time.

10.1.6 Computational results

This section considers some computational experiments to examine the performance of the extended algorithm for mixed set packing/partitioning problems. As indicated above, these types of problems can also be solved if we use slack variables to rewrite them into pure set partitioning problems with the Were we to use this approach, the problem given by [10.1] – [10.4] would be rewritten to:

$$\text{Min} \quad \sum_{j \in J} c_j \cdot x_j \qquad [10.14]$$

Subject to:

$$\sum_{j \in J} a_{rj} \cdot x_j = 1 \qquad \forall r \in R^{SPP} \qquad [10.15]$$

$$\sum_{j \in J} a_{rj} \cdot x_j + s_r = 1 \qquad \forall r \in R^{SPC} \qquad [10.16]$$

$$x_j \in \{0,1\} \qquad \forall j \in J \qquad [10.17]$$

$$s_r \in \{0,1\} \qquad \forall r \in R^{SPC} \qquad [10.18]$$

This computational experiment examines the performance of the algorithm for mixed problems to the performance of LaRSS on the rewritten problems for 500 real-life problems coming from the case discussed in Chapter 8. In this case study, the second approach is used to solve these problems. Table 10.1 shows the characteristics and solution statistics for these 500 problems.

Table 10.1: Computational results mixed set packing/partitioning problems

	Pure set partitioning	Mixed set packing / partitioning
Number of instances	500	500
Average number of constraints	45.08	45.08
Average number of packing constraints	0.00	36.59
Average number of partitioning constraints	45.08	8.49
Average number of variables	10323.07	10286.49
Total solution time	63.77	123.86

There are 500 problems, with an average of 45 constraints, 37 of which are inequality constraints. The mixed problems have an average of 10,286 variables, while the pure set partitioning problems have an average of 10,323 variables. Note

that if |J| is the number of variables in the mixed problem and $|R^{SPC}|$ the number of inequality constraints, then the number of variables in the corresponding pure set partitioning problem is equal to $|J| + |R^{SPC}|$. Using the adapted version of the solver to solve these problems takes approximately twice as long as solving the rewritten problems with LaRSS. This result can be explained by the fact that all methods used are originally designed for set partitioning problems and are altered in order to be able to solve these problems. Moreover, the number of variables of the set partitioning problems is on average only 37 higher than the number of variables in the mixed problems, which is a very small difference. Rewriting the problems into pure set partitioning problems seems to be the most appropriate way to solve this class of problems. The difference in computing time between the two approaches is caused mostly by the lower bound calculation and the branch and bound routine. The subgradient search method takes on average 26% longer in the set packing approach than in the set partitioning approach. The bounds found hardly differ between the two different approaches; the lower bounds found are on average only 0.3% higher with the set partitioning approach, while the upper bounds found are on average 0.5% lower.

Since we have adapted the solver to consider the mixed problems as well, we expect the performance on the pure set partitioning problems to be worse than the performance of LaRSS. Comparing computing times of both solvers on our test set of 58 problems, we find that the adapted solver is only 6% slower than LaRSS.

10.2 Set partitioning with side-constraints

This section examines set partitioning problems with general side-constraints:

$$\text{Min} \quad \sum_{j \in J} c_j \cdot x_j \qquad [10.19]$$

Subject to:

$$\sum_{j \in J} a_{rj} \cdot x_j = 1 \qquad \forall r \in R^{SPP} \qquad [10.20]$$

$$\sum_{j \in J} a_{rj} \cdot x_j \leq 1 \qquad \forall r \in R^{SPC} \qquad [10.21]$$

$$\sum_{j \in J} b_{rj} \cdot x_j = n_r \qquad \forall r \in R^{GE} \qquad [10.22]$$

$$\sum_{j \in J} b_{rj} \cdot x_j \leq n_r \qquad \forall r \in R^{GI} \qquad [10.23]$$

$$x_j \in \{0,1\} \qquad \forall j \in J \qquad [10.24]$$

Here, $R = R^{SPP} \cup R^{SPC} \cup R^{GE} \cup R^{GI}$ is the set of constraints of the problem and J is the set of variables. J and $\{a_{rj}\}$ are defined analogously to the set partitioning problem as discussed in Section 1.1. We define $R^{SP} = R^{SPP} \cup R^{SPC}$ and $R^{G} = R^{GE} \cup R^{GI}$. Furthermore, parameters $\{b_{rj}\}_{j \in J, r \in R^G}$ and $\{n_r\}_{r \in R^G}$ are defined to be nonnegative and

integer. For the techniques to be of use, the set of general constraints R^G is assumed to be relatively small compared to the set of set packing/partitioning constraints R^{SP}.

Sections 10.2.1 to 10.2.5 review all the techniques used to solve set partitioning problems that are discussed in this thesis and examine how these techniques have to be adjusted to solve the more general problems considered here. Section 10.2.6 considers computational experiments of this extended solver.

10.2.1 Preprocessing

Equal columns

If the elements of two columns are equal both R^{SP} and R^G, and have at least one nonzero element in R^{SP} in common, then the column with the highest costs can be deleted. Note that in the implementation of the equal columns preprocessing rule, as discussed in Appendix A, we now use the sum of squared coefficients of the rows in R(j) instead of the row indices as an argument in the sorting of the columns.

Equal rows

If two rows are equal and both rows have the same sign, then one of these rows can be deleted. If two rows have equal indices but different signs, then the row with the inequality sign can be deleted. More formally:

- If $J(r) = J(s)$ for $r,s \in R^{SPP}$, then row s can be removed from the problem.
- If $J(r) = J(s)$ for $r \in R^{SPP}$ and $s \in R^{SPC}$, then row s can be removed from the problem.
- If $J(r) = J(s)$ and $b_{rj} = b_{sj}\ \forall j \in J(r)$ for $r \in R^G, s \in R^{GI}$, then row s can be removed from the problem.
- If $J(r) = J(s)$ and $b_{rj} = b_{sj}\ \forall j \in J(r)$ for $r,s \in R^{GE}$, then row s can be removed from the problem.

Contained rows

The contained rows rule can only be applied to rows in R^{SP}, similar to Section 10.1. Formally:

If $J(r) \subseteq J(s)$ for $r \in R^{SP}$, $s \in R^{SPP}$, then all $j \in J(s)/J(r)$ and s can be removed from the problem.

Clique

If all columns that cover row $r \in R^{SPP}$ have one or more elements in common with a column j that does not cover row r, then we can remove column j, since choosing this column in a solution set will leave constraint r unsatisfiable. There is no straightforward extension of this rule to rows in $R \setminus R^{SPP}$.

Row combination technique

The concept of the row combination technique can be applied effectively only on two set partitioning constraints. In this general setting, we apply the row combination technique only on rows in R^{SPP}.

10.2.2 Lagrangian relaxation and dual heuristics

In order to reformulate the problem, we introduce the following notation:

$$d_{rj} = a_{rj} \qquad \forall r \in R^{SP} \qquad [10.25]$$

$$d_{rj} = b_{rj} \qquad \forall r \in R^{G} \qquad [10.26]$$

$$n_r = 1 \qquad \forall r \in R^{SP} \qquad [10.27]$$

$$R^{I} = R^{SPC} \cup R^{GI} \qquad [10.28]$$

$$R^{E} = R^{SPP} \cup R^{GE} \qquad [10.29]$$

We can now rewrite the problem of [10.19] – [10.24]:

$$\text{Min} \quad \sum_{j \in J} c_j \cdot x_j \qquad [10.30]$$

Subject to:

$$\sum_{j \in J} d_{rj} \cdot x_j = n_r \qquad \forall r \in R^{E} \qquad [10.31]$$

$$\sum_{j \in J} d_{rj} \cdot x_j \le n_r \qquad \forall r \in R^{I} \qquad [10.32]$$

$$x_j \in \{0,1\} \qquad \forall j \in J \qquad [10.33]$$

The Lagrangian relaxation of this problem is given by:

$$z_{LR}(\lambda) = \min \sum_{j \in J} \left(c_j - \sum_{r \in R} d_{rj} \lambda_r \right) \cdot x_j + \sum_{r \in R} n_r \cdot \lambda_r \qquad [10.34]$$

Subject to:

$$\lambda_r \text{ unrestricted} \qquad \forall r \in R^{E} \qquad [10.35]$$

$$\lambda_r \le 0 \qquad \forall r \in R^{I} \qquad [10.36]$$

The dynamic convergent series method has to be adjusted to solve this Lagrangian relaxation problem. The first adjustment concerns the gradient gr^k of [3.30], which is altered to:

$$g_r^k = n_r - \sum_j d_{rj} \cdot x_j^k \qquad r \in R \qquad [10.37]$$

The stepsize, given by [3.31], is changed accordingly. The update function is changed to:

$$\lambda_r^{k+1} = \begin{cases} \min\left(\lambda_r^k + \text{stepsize}^k \cdot g_r^k, 0\right) & \text{if } r \in R^{I} \\ \lambda_r^k + \text{stepsize}^k \cdot g_r^k & \text{else} \end{cases} \qquad [10.38]$$

The Lagrangian costs of a column j is adjusted to be:

$$cl_j = c_j - \sum_{r \in R} d_{rj} \cdot \lambda_r \qquad [10.39]$$

Given a dual feasible vector u^f, the reduced costs of a column are given by:

$$cr_j = c_j - \sum_{r \in R} d_{rj} \cdot u_r^f \qquad [10.40]$$

Given a dual feasible vector u^f and a partial solution x, a lower bound for the total problem is now given by:

$$\sum_{j \in J} cr_j \cdot x_j + \sum_{r \in R} n_r \cdot u_r^f \qquad [10.41]$$

Section 10.2.5 examines how to use the reduced costs in this setting.

Dual improvement heuristic

If for some row r in R^E it is true that the reduced costs of all columns j in J(r), given by [10.40], are strictly positive, then we can raise u_r by:

$$\Delta = \min_{j \in J(r)} cr_j \qquad [10.42]$$

If for some row r in R^I it is true that the reduced costs of all columns j in J(r), given by [10.40], are strictly positive, then we can raise u_r by:

$$\Delta = \min\left(-u_r \ , \ \min_{j \in J(r)} cr_j \right) \qquad [10.43]$$

Dual 3OPT heuristic

Suppose we have a dual feasible vector u_r^f. The objective of the 3OPT heuristic is to find three rows r_1, r_2 and r_3 such that we can decrease u_{r1}^f and increase u_{r2}^f and u_{r3}^f all with the same amount, such that the lower bound is raised. If we are dealing with a set packing/partitioning problem, we can define two simple conditions, [3.39] and [3.40], that have to be fulfilled in order to find r_1, r_2 and r_3. This implies that we do not have to search all row triples, but only those rows for which those conditions hold. In this case, however, we cannot define such conditions beforehand, since the right-hand-side of the constraints can be larger than one. We must thus check every row triple to determine which ones can be used in the 3OPT heuristic. Since this is too time-consuming, we perform the 3OPT heuristic only for the rows in R^{SP}.

10.2.3 Primal heuristic

The concept of the primal heuristic, adding columns in a certain order to a partial solution until we a feasible solution is found, can be applied to this type of problems as well. To this end, we use the framework as described by Figure 4.1. The only adjustment we have to make to this framework is that, in every iteration of the primal heuristic, we consider only those rows with an equality sign to be covered next. If all equality constraints are satisfied, we register the solution. If there is an equality constraint that cannot be satisfied, we move on to the next iteration.

10.2.4 Branch and bound

We use the dynamic constraint-based branching method of Section 5.3.2 and make two adjustments:

- To "cover" a certain row r with an equality constraint and right-hand-side $n_r > 1$, we may need to have more than one column j from J(r). This has to be taken into account in the branching procedure. Obviously, this also holds for inequality constraints with right-hand-side > 1.
- Before we consider the rows with an inequality sign, we first cover all rows with an equality sign in the dynamic sequence.

10.2.5 Other adjustments

The complicating factor of mixed set packing/partitioning problems, discussed in Section 10.1.5, also holds for the inequality constraints in the problems discussed here. Consider the problem formulation of [10.30] – [10.33]. When x is a feasible solution to the mixed set partitioning/packing problem and u_r^f the best dual feasible solution found, the value of the solution is given by:

$$z^{solution} = \sum_{j \in J} cr_j \cdot x_j + \sum_{r \in R} n_r \cdot u_r^f - \sum_{r \in R^I} \left(n_r - \sum_{j \in J(r)} d_{rj} \right) \cdot u_r^f \qquad [10.44]$$

However, the lower bound we have at that point is equal to:

$$z^{LB} = \sum_{j \in J} cr_j \cdot x_j + \sum_{r \in R} n_r \cdot u_r^f \qquad [10.45]$$

In contrast to the set partitioning case, where the lower bound equals the value of the solution when the solution is feasible, here the lower bound is generally lower than the value of the solution. To solve this problem, we correct the solutions found in the primal heuristic and the branch and bound routine for the inequality constraints that are not strictly covered. This has an obvious disadvantage; since the value of the solution is underestimated during the branch and bound routine, we need to go much "deeper" in the branch and bound tree to find out whether the solution is indeed better than the upper bound known at that time.

10.2.6 Computational results

The first computational experiment we perform considers the case study discussed in Chapter 8. As mentioned in Section 8.5, this case can be solved more easily if general side-constraints can be handled. Especially when the number of must-orders exceeds the number of vehicle-day combinations, the pure set partitioning approach as given by [8.6] – [8.12] appears to work poorly and much progress can be achieved by allowing side-constraints. To illustrate this, we examine 750 instances of the validation scenarios where we assume only one vehicle to be available instead of

two. We choose these scenarios since the number of vehicle-day combinations is smaller and has to be taken into account in the planning problem. These scenarios can be solved as pure set partitioning instances, as mixed set packing and partitioning instances and as set partitioning instances with side-constraints. Table 10.2 shows the results of the three approaches. Note that in this table, the 'average time in solver optimal' denotes the average time on the 739 instances that are solved to optimality with all three approaches.

Table 10.2: Results of the three different approaches

Number of instances	750
Average number of variables	1428
Average number of constraints	20
Average number of packing constraints	14
Average number of general side-constraints	1
Average time LaRSS	5.08
Average time LaRSS optimal	0.62
Number of times not solved in 5 minutes LaRSS	11
Average time in packing solver	3.98
Average time in packing solver optimal	0.20
Number of times not solved in 5 minutes packing	6
Average time in plus solver	0.47
Average time in plus solver optimal	0.03
Number of times not solved in 5 minutes plus	0

As can be seen, the partitioning approach performs the worst, with 11 instances not solved within five minutes and an average time of 0.62 seconds on the remaining instances. The "plus" approach, with the solver that handles side-constraints, works the best, where all instances are solved well within five minutes, with an average time of 0.47 seconds and only 0.03 seconds on the 739 instances that are solved within five minutes in all three approaches. These results illustrate the value of this extended solver in this real-life case study. Obviously, we cannot generalize these results, since the problems considered here are relatively small.

Since we have no test set of set partitioning problems with side-constraints available, but we want to examine further the performance of the extended solver, we take four set partitioning problems out of our test set and randomly add some side-constraints. To this end, we take four problems with a moderate number of columns and rows in our test set: kl02, nw17, us03 and us04. For every problem we create 12 new instances, such that a total of 48 set partitioning problems with side-constraints are available. This is done in the following way:

- The number of added constraints is m% of the original number of constraints, where m equals 5, 10, 15, 20, 25 and 30 respectively.

- The number of nonzero's is uniformly distributed between n_1% and n_2% of the original number of columns, where (n_1, n_2) is equal to (1,2) in the first six cases and (2,4) in the last six cases.
- The right-hand side of every constraint is drawn from the uniform(10,25) distribution.
- The value of every coefficient is drawn from the uniform(2,5) distribution.
- The sign of all side-constraints is "≤".

Table 10.3 shows the results of the extended solver compared to the results of CPLEX on these 48 instances. In 36 cases, CPLEX performs worse than our extended solver, while for the remaining 12 instances there are two where our solver is not finished within ten minutes. Note that all 12 instances stem from the original set partitioning instance kl02. Excluding the two instances that are not solved within ten minutes, the average time of the extended solver is 12 seconds, versus 22 seconds for CPLEX. These results give a mixed image of the performance of the solver: although in most cases the solver performs very well, even twice as fast as CPLEX, there are problem structures that cannot be handled very well. The performance of CPLEX may be worse on most instances; it seems to be more robust. Note that this conclusion is based on a relatively small test set of randomly generated instances; further research is thus our recommendation.

Table 10.3: Performance of the extended solver 48 test problems

Number of instances	48
Average number of variables	67218.50
Average number of constraints	109.88
Average number of side-constraints	16.88
Average number of nonzero's side-constraints	1535.27
Number of instances time set partitioning solver > 10 min	2
Number of instances time CPLEX > 10 min	0
Average time set partitioning solver remaining instances	11.55
Average time CPLEX remaining instances	21.88
Number of times set partitioning solver faster	36
Number of times CPLEX faster	12

Since we have adapted the solver to consider the general problems as well, we expect the performance on the pure set partitioning problems to be worse than the performance of LaRSS. Comparing computing times of both solvers on our test set of 58 problems, we find that the adapted solver is 33% slower than LaRSS.

10.3 Concluding remarks

This chapter examined two possible extensions of LaRSS. The first extension considers mixed set packing/partitioning problems. We discussed how the methods we designed for solving set partitioning problems can be extended for this more general problem class and considered some computational experiments. When problem sizes are relatively small, this extension of the solver can be useful in solving mixed set packing/partitioning problems more quickly. When problem sizes are larger, however, it appears to be faster to use slack variables to rewrite the problem to a pure set partitioning problem.

The second extension of the solver considers set partitioning problems with more general side-constraints. Again, we examined how the techniques designed for pure set partitioning problems can be adjusted to be able to cope with this type of problems. Two computational experiments are conducted. The first experiment indicates that this extended solver is very useful in reducing the solution times of the problems we considered in the case described in Chapter 8. In this case, we need a trick to rewrite the encountered problem, a set partitioning problem with side-constraints, to a pure set partitioning problem. This trick involves adding a large number of extra variables and creating a problem with many solution with the same solution value. Solving these problems with LaRSS takes much more time than solving the original problems with the extended solver. The second experiment considers pure set partitioning problems from our test set with randomly added side-constraints. The results of the extended solver on these 48 problems indicate that the solver is faster than CPLEX on most of the problems, although it is much slower on a small subset of the problems. Although the performance of the extended solver is promising, it does not seem to be very robust.

In general, for both extended solvers, we can say that the developed techniques work better with equality constraints than with inequality constraints. This is caused by the extra adjustments needed to deal with the reduced costs, as discussed in Sections 10.1.5 and 10.2.5 and is illustrated by the results of the computational experiments on the mixed packing / partitioning problems. These problems can easily be rewritten to pure set partitioning problems, while this cannot de done straightforwardly for the general problems discussed in Section 10.2. For both extensions we recommend further research to discover the potential of the methods described. In particular we recommend more extensive computational experiments on real-life set partitioning problems with general side-constraints.

Chapter 11

Epilogue

This thesis discusses the development of LaRSS; the Lagrangian relaxation-based solver for set partitioning problems. Section 11.1 summarizes the findings of the thesis. Section 11.2 relates the results of the thesis to the goal of the research and lists the scientific contributions of our research. Finally, Section 11.3 gives some recommendations for further research.

11.1 Summary

Given a collection of subsets of a certain root set and costs associated to these subsets, the set partitioning problem is the problem of finding a minimal cost partition of the root set. Many real-life problems can be formulated as a set partitioning problem, where the most important applications lie in the fields of vehicle routing and crew scheduling. In this thesis we discuss the development of LaRSS, a Lagrangian relaxation based set partitioning solver, that can be used to solve set partitioning problems to optimality fast. In literature much attention has been given to solving set partitioning problems, however, all successful algorithms discussed in literature are based on well developed linear programming software, mostly CPLEX. The purpose of our research is to achieve a good performance, measured in computing time, without using external solvers to solve the relaxations.

11.1.1 Preprocessing

One of the most important building blocks of LaRSS is the set of preprocessing methods that is discussed in Chapter 2. The aim of preprocessing is to reduce the solution time of a set partitioning problem by reducing the number of rows and/or the number of columns of the problem. To this end, five techniques are

implemented in LaRSS: equal columns, contained rows, cliques, equal rows and the row combination technique. The first four are known from the set partitioning literature. The row combination technique is an extraordinary preprocessing technique, since it allows for a small increase in the number of columns in order to realize a reduction in the number of rows and therewith a reduction in the solution time of the problem. Another newly developed preprocessing rule, the cut rule, turns out to be of little value. In the computational experiments, preprocessing proves to be of great value to reduce the solution time of set partitioning problems.

11.1.2 Lower bounds

The quality and computation time of the lower bounds are of great importance to the performance of LaRSS. As discussed in Chapter 3, we have implemented and analyzed several subgradient search algorithms to solve the Lagrangian relaxation of the set partitioning problem. We introduce three new subgradient search methods that are based on the the convergent series method of Goffin (1977): the static convergent series method, the dynamic convergent series method and the bundle convergent series method. The convergent series method is applied rarely in practice, while the methods we examined work very well for the set partitioning problem. The dynamic convergent series method we implement in LaRSS outperforms other well-known techniques like the classic subgradient search method of Held, Wolfe and Crowder (1974) and the volume algorithm of Barahona and Anbil (2000, 2002). Moreover, the methods implemented in LaRSS allow us to find lower bounds close to and sometimes even better than the LP lower bounds, while the computing times are comparable.

11.1.3 Upper bounds

Although the knowledge of an upper bound is not essential for the branch and bound algorithm of LaRSS to work, it does have a positive effect on the performance of the algorithm. Therefore, we perform a simple greedy primal heuristic to try to find an upper bound fast. This heuristic is the subject of Chapter 4.

11.1.4 Branch and bound

After performing preprocessing techniques and the search for lower- and upper bounds, LaRSS finds the optimal solution with a branch and bound algorithm. We have examined several branching strategies; variable-based versus constraint-based and static versus dynamic, and implemented a dynamic constraint-based procedure in LaRSS. Chapter 5 discusses the theory and results of the research leading to this algorithm. From our test set of 60 problems, two are not solved satisfactorily by this procedure. To improve the performance of the algorithm, we

examine the effect of using dual heuristics and Lagrangian relaxation during branch and bound. We conclude that these extra calculations can have a positive effect on the performance on the difficult problems, but for most problems the extra time needed is not offset by the benefit of the better lower bounds.

11.1.5 LaRSS

Chapter 7 examines the construction of LaRSS and its performance on our test set of 60 problems. Apart from the two difficult instances, LaRSS performs very well: the total computing time on the 58 instances is only 174 seconds against 1,070 seconds for CPLEX. Including the two difficult instances, LaRSS performs better on 77% of the problems. The two difficult instances, however, are not solved by LaRSS wihin 24 hours, while CPLEX solves them within minutes. For these instances we lack the knowledge of good lower bounds during branch and bound in order to close the gap from the dual side. We will come back to this issue in the conclusion in Section 12.1.1.

11.1.6 Case studies

Chapters 8 and 9 discuss two cases that are performed for Auto Recycling Nederland, where LaRSS is used to solve the set partitioning problems encountered in the planning processes. In the case of Chapter 8, where we consider the collection of fluids coming from end-of-life vehicles, we encountered quite small set partitioning problems. In total we examine the performance on 14,172 problems, from which 15 are not solved to optimality in 10 minutes. The average computing time of LaRSS on the remaining 14,157 problems is 0.6 seconds. We have performed some more research on the 15 difficult instances. From these problems, 3 are also not solved to optimality in 10 minutes by CPLEX. The average time of CPLEX on the remaining 12 problems is equal to 98 seconds. The problems considered in this case are actually set partitioning problems with a small number of side-constraints, that are rewritten to pure set partitioning problems. The extended solver of Section 10.2, that can handle set partitioning problems with side-constraints, does solve these 15 problems in an average time of 2.0 seconds.

The case of Chapter 9 examines the collection of containers from end-of-life vehicle dismantlers and the transport to recyclers by logistics service providers. Several scenario's are considered, where set partitioning is used to solve the operational planning problem. For the problems considered in this case we have made a comparison between the performance of LaRSS and CPLEX. The size of the problems as well as the performance of the solvers depend very much on the type of scenario. In scenario's where many solutions exist with the same or very close solution values, LaRSS tends to perform worse than CPLEX, while in problems with larger cost differences, LaRSS performs better than CPLEX, even though these

problems are much larger in terms of number of variables and constraints. On the total set of 6,360 problems, CPLEX performs better on average and is more robust.

11.1.7 Extensions

Chapter 10 considers two extensions of LaRSS: mixed set packing / partitioning problems and set partitioning problems with general side-constraints. With the use of extensive testing on a set of real-life problems coming from the case of Chapter 8, we found that mixed set packing / partitioning problems are solved faster by using slack variables to rewrite the problem to pure set partitioning problems. The extended solver aimed at solving set partitioning problems with general side constraints, on the other hand, is indeed successful in solving this type of problems. The 15 problems of the case study of Chapter 8 that are not solved to optimality in 10 minutes are solved by this extended solver in and average time of 2.0 seconds if they are not rewritten to pure set partitioning problems. Moreover, on 48 set partitioning problems with randomly generated side constraints, the extended solver performs good: in 36 of the 48 instances, the solver is faster than CPLEX. Note that all problems considered either contain a very small amount of side-constraints, or only randomly generated side-constraints. We recommend more research on the performance of this extended solver on real-life set partitioning problems with a larger amount of side-constraints.

11.2 Conclusion

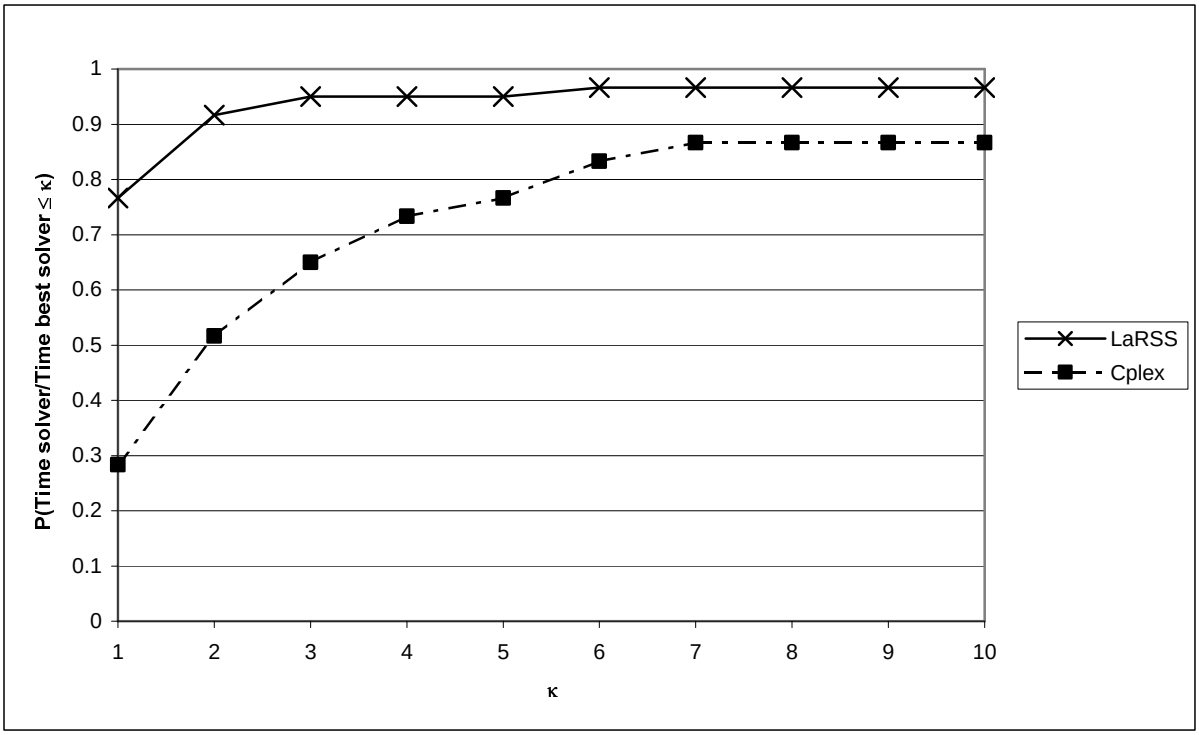
This section concludes the research presented in this thesis. First, we discuss the performance of LaRSS and relate our findings to the goal and motivation of this research as discussed in Chapter 1. Next, we list the most important contributions of our research.

11.2.1 Conclusion on the performance of LaRSS

The goal of the research presented in this thesis was formulated in Chapter 1 to find out if a Lagrangian relaxation based branch and bound algorithm, without using any external mathematical programming solvers, can achieve the same kind of performance as the successful linear programming based algorithms that are described in the literature in the last decades. In our opinion, the most important reference points are the algorithm of Hoffman and Padberg (1993), the work of Borndörfer (1998) and off course the mathematical programming solver CPLEX itself. Like most researchers in the field, we use CPLEX as our benchmark, since this is a well developed and widely available solver. The other two mentioned solvers were not available to the author for comparison.

The most important test set we used in this thesis is the set of problems coming from the OR Library that is used in many of the papers on solving set partitioning problems. This set consists of 60 problems. Figure 11.1 shows the performance profile of LaRSS and CPLEX for this set of problems. Note that, as indicated in Chapter 7, a performance profile shows for both solvers the probability, based on the current test set, that the computing time of the solver is within a factor κ times the computing time of the best solver. For example, Figure 11.1 indicates that in 92% of the instances the computing time of LaRSS is within a factor two of the time of the best solver for 95% of the problems.

Figure 11.1: Performance profile of LaRSS versus CPLEX on the academic test set



The performance profile illustrates the good performance of LaRSS on this test set, since the performance line of LaRSS is above the performance line of CPLEX. However, two problems of the set are not solved by LaRSS in a reasonable amount of time, not even in 24 hours. The lines in the performance profile will thus cross at some point κ. Leaving the problems that are solved in less than 0.001 seconds by LaRSS out of account, this point will be at κ equal to 108. For one of the puzzle instances, the Heart puzzle, LaRSS is even 649 times as fast as CPLEX. On average, LaRSS is 25 times as fast CPLEX on the five puzzle instances. Since these problems are feasibility problems, knowledge of the LP solution during branch and bound gives no advantage and recalculating the LP relaxation on every node is useless. This gives LaRSS an advantage over CPLEX for these instances.

The two problems not solved in 24 hours are noted in the literature to be hard,

although the known LP-based algorithms are able to solve these problems in a reasonable amount of time. CPLEX solves the problems in less than two minutes. Borndörfer (1998) reports solution times of 342 seconds for aa04 and 214 seconds for aa01. Hoffman and Padberg (1993) report computing times of 4.0 hours for aa01 and 38.7 hours for aa04. The difficulty of these problems is not caused by simple characteristics as the number of rows or columns or the density of the constraints matrix, but lies in the complicated cost structures of the problems. Since cost differences are small and many solutions exist with comparable costs, there are too many options to be checked during the branch and bound procedure. This would be easier to handle if we had better knowledge of lower bounds during the branch and bound, since then we could cut off branches much earlier in the process. This observation is also illustrated by the fact that one of the difficult problems can be solved in 41 minutes if subgradient search is used during branch and bound to update the lower bounds. This is the great disadvantage of Lagrangian relaxation based methods compared to linear programming based methods: when a solution to the LP relaxation is known, the updating of the lower bound in the branch and bound tree is very simple and can be done very quickly. However, when performing a subgradient search on a certain node in the branch and bound tree, the process has to be started all over again. For the problem that is solved in 41 minutes, over 3,400 runs of the subgradient search algorithm have to be performed. Realizing this, the total time of 41 minutes is not even that long. An advantage of recalculating the LP problem during branch and bound is that infeasibility of the induced subproblem can be determined on every node of the branching tree, such that the nodes can be fathomed much faster.

Another set of problems for which a comparison with CPLEX is made in this thesis comes from the real-life case study discussed in Chapter 9. Figure 11.2 shows the performance profile of LaRSS versus CPLEX for this set of problems. This profile again shows the good performance of LaRSS: in 69% of the cases LaRSS is the fastest of the two solvers. However, we see that there is a crossing in this picture, around $\kappa = 8$, meaning that LaRSS has more problems with relatively large calculating time than CPLEX. In other words: CPLEX is more robust in solving this set of problems, although LaRSS is faster than CPLEX in most cases.

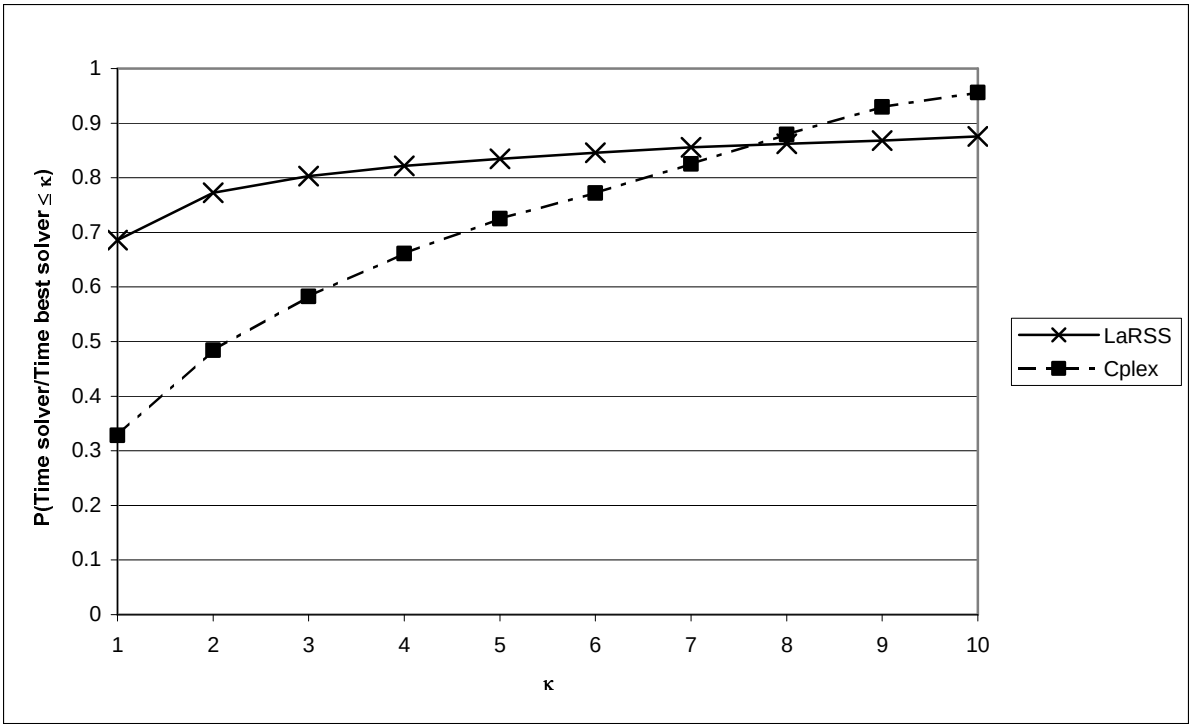
In conclusion we can say that indeed LaRSS is powerful in solving set partitioning problems. The power of the algorithm lies primarily in the problem reduction techniques and the subgradient search algorithm. The problem reduction heuristics implemented are very fast and successful in reducing the size of the problem. The subgradient search algorithm that is implemented to solve the Lagrangian relaxation problem gives very high quality lower bounds fast and can compete with linear programming relaxations in terms of time as well as quality.

In general we can say that LaRSS is faster than CPLEX on the majority of the set partitioning problems. CPLEX and other linear programming based algorithms, however, are more robust in solving set partitioning problems, meaning that there are

less problems for which the computing times are extremely high. For problems with difficult cost structures, LP-based techniques have three important advantages over Lagrangian relaxation based algorithms like LaRSS:

- The solution to the linear programming relaxation can be updated very easily on every node to improve the lower bound.
- Since the solution to the linear programming relaxation is updated on every node, infeasibility of the induced subproblem can be determined instantaneously.
- The solution to the linear programming relaxation can be used to develop and evaluate cuts.

Figure 11.2: Performance profile of LaRSS versus CPLEX on the case study test set



11.2.2 Contributions

This section summarizes the most important scientific contributions of this thesis.

- Evaluation of known preprocessing techniques including extensive computational experiments and a discussion of implementations.
- Development and implementation of the row combination technique; a powerful preprocessing technique for set partitioning problems.
- Development and implementation of a new subgradient search method based on the convergent series method of Goffin (1977). This method finds good lower bounds for set partitioning problems quickly and outperforms other well-known subgradient search methods of Held et al. (1974) and Barahona and Anbil (2000, 2002). Moreover, the performance on the Lagrangian relaxation of set partitioning

problems is comparable to the performance of CPLEX on the LP relaxation of set partitioning problems in terms of both computing time and quality of the bounds.
- Development of a new primal heuristic and a new dynamic constraint-based branch and bound algorithm to solve set partitioning problems without using the solution to the linear programming relaxation.
- Construction of LaRSS, which performs good and better than CPLEX on the majority of the set partitioning problems.
- Adding five set partitioning instances with interesting characteristics to the OR library of Beasley (OR-library, 2004) for research purposes. The instances originate from puzzles, formulated as set partitioning problems, where the goal is to find a feasible solution to the puzzle.
- Showing the value of operations research techniques in general and set partitioning solvers in particular by applying LaRSS successfully in two real-life applications.

Some comments are in place considering the limitations of our research:
- Since LaRSS is a special purpose solver, one would expect it to perform better on set partitioning problems than a general mathematical programming solver like CPLEX. Although this is indeed the case for the majority of the set partitioning problems considered, CPLEX performs more robustly on the whole set. We must, however, realize that CPLEX has a long history of development and also uses problem-recognition techniques and special-purpose routines to consider certain types of problems, including set partitioning problems.
- Our results show that, although the dynamic convergent series method for solving the Lagrangian relaxation performs very well, this does not necessarily imply that it can be used successfully in a solution algorithm. When comparing the method to the performance of CPLEX on the LP relaxation, we note that the bounds and computing times are comparable, however the LP-based methods have many more possibilities when applied in a solution algorithm.
- Although our results show that LaRSS outperforms CPLEX on the majority of the set partitioning problems, we have not been able to identify the characteristics of these problems. We believe that there are applications of set partitioning problems where this advantage can be exploited, nevertheless we did not succeed in finding such applications.

11.3 Recommendations

An important result of this research concerns the five puzzle instances, introduced in Section 1.5. The performance of LaRSS on these instances is much better than the performance of CPLEX: on average, CPLEX needs 25 times the computing time of LaRSS on these instances. The puzzle instances are in fact feasibility problems,

where the bounds have no impact on the performance of the solver. The good performance of LaRSS on these instances indicate that the solver might be used successfully to solve logical problems. Research on the connection between logic and optimization is relatively new and can work in both ways, see for example Hooker (2000). We recommend more research on the possible value of mathematical programming solvers in the field of logical problems. LaRRS can play an interesting role, considering the good performance on the puzzle instances.

More research is recommended to establish the value of the extended version of LaRSS for set partitioning problems with side-constraints. We showed that when the side-constraints consist of set packing constraints only, the problem is best solved by rewriting it into a pure set partitioning problem and using LaRSS. However, when more general side-constraints exist, rewriting is not done straightforwardly. Tests indicate that the extended solver of Section 10.2 can be used successfully for set partitioning problems with general side-constraints, especially when the side-constraints are equality constraints. More extensive testing with real-life problems is needed to confirm this preliminary result. Our expectation is, however, that the same conclusion will hold for the extended version of LaRSS as does for LaRSS itself: the performance on a large set of problems might be very good, however the linear programming-based methods will perform more robustly on the whole set of problems.

Although LaRSS is successful in solving most set partitioning problems to optimality quickly, linear programming based methods are more robust on the whole class of set partitioning problems. Which solver to use in a particular situation depends on the goal and preferences of the user and cannot be prescribed by a straightforward recommendation. Large scale companies that are faced with practical set partitioning problems daily, are probably best off with the CPLEX solver, since it is more robust, more generally applicable and has a long history of development. For small projects and for research purposes, LaRSS has advantages over ‘black box’ solvers like CPLEX, since it is easy to incorporate special knowledge about the problems considered and LaRSS has been proven to perform very well on set partitioning problems.

Appendix A

Preprocessing implementations

In this appendix, we describe the implementations of the preprocessing techniques discussed in Chapter 2.

A.1 Equal columns

Comparing all elements for every pair of columns would be very time-consuming. To prevent this, we use a pretest to determine whether a full check is worthwhile.

Figure A1: The implementation of the equal columns preprocessing rule

```
FOR (j ∈ J)
    test_var[j] = Σ_{r∈R(j)} r²
ENDFOR
Sort columns on increasing value of test_var[j]
FOR (j1 ∈ J)
    FOR (j2 ∈ {j1 + 1, ..., |J|})
        IF ((|R(j1)| = |R(j2)|) AND (test_var[j1] = test_var[j2]))
            IF (R(j1) = R(j2))
                IF (c_j1 ≤ c_j2)
                    Delete j2
                ELSE
                    Delete j1
                ENDIF
            ENDIF
        ENDIF
    ENDFOR
ENDFOR
```

A.2 Equal 2-columns

Comparing all elements for every set of three columns would be too time-consuming. Like in the equal columns implementation, we use a pretest to determine whether a full check of a certain set of three columns is worthwhile. The implementation resembles the one for the equal columns rule.

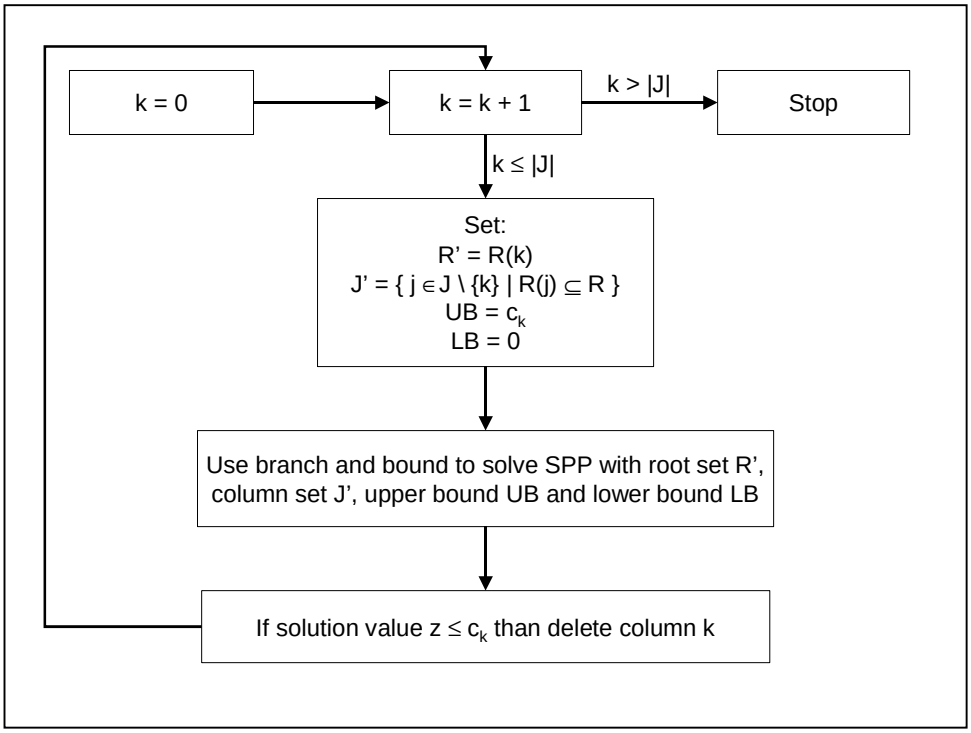
Figure A2: The implementation of the equal 2-columns preprocessing rule

```
FOR(j ∈ J)
    test_var[j] = Σ_{r∈R(j)} r²
ENDFOR
Sort columns on increasing value of test_var[j]
FOR(j3 = 1,...,|J|)
    FOR(j1 = j3 − 1,...,1)
        IF(test_var[j1] ≤ (1/2) test_var[j3])
            break
        ENDIF
        Δ = test_var[j3] − test_var[j1]
        IF(Δ = 0)
            IF(R(j1) = R(j2))
                IF(c_j1 ≤ c_j2)
                    Delete j2
                ELSE
                    Delete j1
                ENDIF
            ENDIF
        ELSE
            FOR(j2 = 1,...,j1 − 1)
                IF(test_var[j2] = Δ)
                    IF((|R(j1)| + |R(j2)| = |R(j3)|) AND (c_j1 + c_j2 ≤ c_j3))
                        IF((R(j1) ∪ R(j2) = R(j3)) AND (R(j1) ∩ R(j2) = ∅))
                            Delete j3
                        ENDIF
                    ENDIF
                ENDIF
            ENDFOR
        ENDIF
    ENDFOR
ENDFOR
```

A.3 Equal k-columns

To perform the equal k-columns rule, we use the branch and bound procedure discussed in Chapter 5 to solve |J| branch and bound problems. For every column k, we formulate a small set partitioning problem consisting of the rows in R(k) and the columns that are contained in column k. If the solution of this set partitioning problem is smaller than c_k, this means that column k can be represented by a set of other columns with lower costs. Figure A.3 shows an outline of the equal k-columns algorithm.

Figure A3: An outline of implementation of the equal k-columns preprocessing rule



A.4 Equal rows

The implementation of equal rows resembles the implementation for equal columns and uses a similar pretest. The algorithm is shown in Figure A.4.

Figure A.4: The implementation of the equal rows preprocessing rule

```
FOR(r ∈ R)
    test_var[r] = Σ_{j∈J(r)} j^2
ENDFOR
Sort rows on increasing value of test_var[r]
FOR(r1 ∈ R)
    FOR(r2 ∈ {r1 + 1,...,|R|})
        IF((|J(r1)| = |J(r2)|) AND (test_var[r1] = test_var[r2]))
            IF(J(r1) = J(r2))
                Delete r2
            ENDIF
        ENDIF
    ENDFOR
ENDFOR
```

A.5 Contained rows

The implementation of the contained rows preprocessing rule is a very straightforward check for every pair of rows.

Figure A.5: The implementation of the contained rows preprocessing rule

```
FOR(r1 = 1,...,|R|)
    FOR(r2 = r1 + 1,...,|R|)
        IF(|J(r1)| ≤ |J(r2)|)
            IF(J(r1) ⊆ J(r2))
                Delete r2
                Delete all j ∈ J(r2) \ J(r1)
            ENDIF
        ELSE
            IF(J(r2) ⊂ J(r1))
                Delete r1
                Delete all j ∈ J(r1) \ J(r2)
            ENDIF
        ENDIF
    ENDFOR
ENDFOR
```

A.6 Equal 3-rowsets

The implementation for equal 3-rowsets incorporates an easy check for contained rows.

Figure A.6: The implementation of the equal 3-rowsets preprocessing rule

```
FOR (r_1 = 1,...,|R|)
    j_1 = argmin_{j∈J(r_1)} |R(j)|
    FOR (r_2 ∈ R(j_1), r_2 ≠ r_1)
        Col_Found = FALSE
        FOR (j_2 ∈ J(r_1) \ {j_1})
            IF (r_2 ∉ R(j_2))
                Col_Found = TRUE
                Break
            ENDIF
        ENDFOR
        IF (Col_Found)
            FOR (r_3 ∈ R(j_2))
                TRIPLET = TRUE
                FOR (j ∈ J(r_1) \ {j_1})
                    IF ((j ∉ J(r_2)) AND (j ∉ J(r_3)))
                        TRIPLET = FALSE
                    ENDIF
                ENDFOR
                IF (TRIPLET)
                    Delete {k ∈ J | k ∉ J(r_1), k ∈ J(r_2), k ∈ J(r_3)}
                ENDIF
            ENDFOR
        ELSE
            Delete r_2
            Delete all j ∈ J(r_2) \ J(r_1)
        ENDIF
    ENDFOR
ENDFOR
```

A.7 Clique

Performing the clique rule takes a relatively long computing time, since we have to do a check for every column.

Figure A.7: The implementation of the clique preprocessing rule

```
FOR(j1 ∈ J)
    FOR(r1 ∈ R\R(j1))
        Reduction = TRUE
        FOR(j2 ∈ J(r1))
            Empty = TRUE
            FOR(r2 ∈ R(j2))
                IF(r2 ∈ R(j1))
                    Empty = FALSE
                    Break
                ENDIF
            ENDFOR
            IF(Empty)
                Reduction = FALSE
                Break
            ENDIF
        ENDFOR
        IF(Reduction)
            Delete j1
            Break
        ENDIF
    ENDFOR
ENDFOR
```

A.8 Cut

Figure A.8: The implementation of the cut preprocessing rule

```
FOR(cut ∈ R)
    next_cut = FALSE
    j_1 = argmin_{j∈J(cut)} |R(j)|
        FOR(r_1 ∈ R(j_1) \ {cut})
            FOR(r_2 ∈ R(j_1) \ {cut}, r_2 > r_1)
                next_column = -1
                FOR(j_2 ∈ J(cut) \ {j_1})
                    IF((r_1 ∉ R(j_2)) AND (r_2 ∉ R(j_2)))
                        next_cut = TRUE
                        Break
                    ENDIF
                    IF(((r_1 ∈ R(j_2)) AND (r_2 ∉ R(j_2))) OR ((r_1 ∉ R(j_2)) AND (r_2 ∈ R(j_2))))
                        next_column = j_2
                        Break
                    ENDIF
                ENDFOR
                IF(next_cut)
                    Break
                ENDIF
                IF(next_column < 0)
                    FOR(r_3 ∈ R \ {r_1, r_2, cut})
                        Delete {k ∈ J | k ∉ J(cut), R(k) ∩ {r_1, r_2, r_3} ≥ 2}
                        next_cut = TRUE
                    ENDFOR
                ELSE
                    FOR(r_3 ∈ R(next_column) \ {r_1, r_2, cut})
                        success = TRUE
                        FOR(j_3 ∈ J(cut) \ {j_1}, j_3 > j_2)
                            IF(R(j_3) ∩ {r_1, r_2, r_3} < 2)
                                success = FALSE
                            ENDIF
                        ENDFOR
                        IF(success)
                            Delete {k ∈ J | k ∉ J(cut), R(k) ∩ {r_1, r_2, r_3} ≥ 2}
                            next_cut = TRUE
                        ENDIF
                    ENDFOR
                ENDIF
                IF(next_cut)
                    Break
                ENDIF
            ENDFOR
        ENDFOR
    ENDFOR
ENDFOR
```

References

Agarwal, Y., Mathur, K. and Salkin, H.M. (1989). A set-partitioning-based exact algorithm for the vehicle routing problem. *Networks,* 19:731-749.

Albers, S. (1980). Implicit enumeration algorithms for the set-partitioning problem. *OR Spektrum,* 2: 23-32.

Archetti, C. and Speranza, M.G. (2004). Vehicle routing in the 1-skip collection problem. *Journal of the Operational Research Society,* 55:717-727.

Atamtürk, A., Nemhauser, G.L. and Savelsbergh, M.W.P. (1995). A combined Lagrangian, linear programming, and implication heuristic for large-scale set partitioning problems. *Journal of Heuristics*, 1: 247-259.

Balas, E. (1977). Some valid inequalities for the set partitioning problems. *Annals of Discrete Mathematics* 1: 13-47.

Balas, E. and Padberg, M.W. (1976). Set partitioning: A survey. *SIAM Review,* 18: 710-760.

Baldacci, R., Hadjiconstantinou, E., Maniezzo, V. and Mingozzi, A. (2002). A new method for solving capacitated location problems based on a set partitioning approach. *Computers & Operations Research*, 29: 365-386.

Barahona, F. and Anbil, R. (2000). The volume algorithm: producing primal solutions with a subgradient method. *Mathematical Programming*, 87: 385-399.

Barahona, F. and Anbil, R. (2002). On some difficult linear programs coming from set partitioning. *Discrete Applied Mathematics*, 118: 3-11.

Beasley, J.E. and Chao, B. (1996). A tree search algorithm for the crew scheduling problem. *European Journal of Operational Research,* 94: 517-526.

Bell, W.J., Dalberto, L.M., Fisher, M.L., Greenfield, A.J., Jaikumar, R., Kedia, P., Mack, R.G. and Prutzman, P.J. (1983). Improving the distribution of industrial gases with an on-line computerized routing and scheduling optimizer. *Interfaces*, 13: 4-23.

Beullens, P. (2001). *Location, process selection and vehicle routing models for reverse logistics*. Dissertation, University of Leuven, Belgium.

Bodin, L., Golden, B., Assad, A. and Ball, M. (1983). Routing and scheduling of vehicles and crews: the state of the art. *Computers and Operations Research*, 10:63-211.

Bodin, L., Mingozzi, A. and Baldacci, R. (2000). The rollon-rolloff vehicle routing problem. *Transportation Science,* 43(3):271-288.

Borndörfer, R. (1998). *Aspects of set packing, partitioning and covering*. Dissertation, Technical University of Berlin, Germany.

Borndörfer, R., Grötschel, M. and Löbel, A. (1998). Optimization of transportation systems. Konrad-Zuse-Zentrum für Informationstechnik Berlin, preprint SC 98-09, Germany.

Butchers, E.R., Day, P.R., Goldie, A.P., Miller, S., Meyer, J.A., Ryan, D.M., Scott, A.C. and Wallace, C.A. (2001). Optimized crew scheduling at Air New Zealand. *Interfaces*, 31:30-56.

Byun, C. (2001). *Lower bounds for large-scale set partitioning problems.* Master Thesis, Technical University of Berlin, Germany.

Campbell, A., Clarke, L. and Savelsbergh, M. (2002). Inventory Routing in Practice. In Toth, P. and Vigo, D. (eds.) *The Vehicle Routing Problem.* SIAM monographs on discrete mathematics and applications, Philadelphia, USA.

Cattrysse, D.G., Salomon, M. and Van Wassenhove, L.N. (1994). A set partitioning heuristic for the generalized assignment problem. *European Journal of Operational Research*, 72:167-174.

Chu, P.C. and Beasley, J.E. (1998). Constraint handling in genetic algorithms: The set partitioning problem. *Journal of Heuristics,* 11: 323-357.

Clarke, G. and Wright, J.W. (1964). Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12: 568-581.

Crowder, H. (1976). Computational improvements for subgradient optimization. *Symposia Mathematica,* 19: 357-372.

De Koster, M.B.M., Flapper, S.D.P., Krikke, H.R. and Vermeulen, W.S. (2005). Recovering end-of-life large white goods: the Dutch initiative. In Flapper, S.D.P., Van Nunen, J.A.E.E. and Van Wassenhove, L.N. (eds.) *Managing closed-loop supply chains*. Springer Verlag, Berlin, Germany.

De Meulemeester, L., Laporte, G., Louveaux, F.V. and Semet, F. (1997). Optimal sequencing of skip collections and deliveries*. Journal of Operational Research Society,* 48:57-64.

Desaulniers, G., Desrosiers, J., Dumas, Y., Marc, S., Rioux, B., Solomon, M.M. and Soumis, F. (1997). Crew pairing at Air France. *European Journal of Operational Research*, 97:245-259.

Dethloff, J. (2001). Vehicle routing and reverse logistics: the vehicle routing problem with simultaneous delivery and pick-up. *OR Spectrum,* 23:79-96.

Directive 2000/53/EC of the European Parliament and of the council of 18 September 2000 on end-of-life vehicles. *Official Journal of the European Communities* L 269: 34-42.

Dolan, E.D. and Moré, J.J. (2002). Benchmarking optimization software with performance profiles. *Mathematical Programming* 91 (2): 201-213.

Dror, M. and Ball, M. (1987). Inventory/Routing: Reduction from an annual to a short-period problem. *Naval Research Logistics* 34: 891-905.

Dror, M. and Levy, L. (1986). A vehicle routing improvement algorithm comparison of a greedy and a matching implementation for inventory routing. *Computers & Operations Research* 3:1: 33-45.

Eternity (2004). Description of the Eternity puzzle. http://www.eternity-puzzle.co.uk.

Evo-IT (2004). IT company in the Netherlands. http://www.evo-it.nl.

Exotic (2004). Description of the Exotic Fives puzzle. http://www.puzzles.force9.co.uk/gall2/exotic5.htm.

Falkner, J.C. and Ryan, D.M. (1987). A bus crew scheduling system using a set partitioning model. *Asia-Pacific Journal of Operational Research*, 4:39-56.

Fisher, M.L. (1981). The Lagrangian relaxation method for solving integer programming problems. *Management Science,* 27: 1-18.

Fisher, M.L. and Kedia, P. (1990). Optimal solutions of set covering/partitioning problems using dual heuristics. *Management Science,* 36: 674-688.

Fleuren, H.A. (1988). *A computational study of the set partitioning approach for vehicle routing and scheduling problems*. Dissertation, University of Twente, The Netherlands.

Foster, B.A. and Ryan, D.M. (1976). An integer programming approach to the vehicle scheduling problem. *Operational Research Quarterly*, 27: 367-384.

Garey, M.R. and Johnson, D.S. (1979). *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman, New York.

Geoffrion, A.M. (1974). Lagrangian relaxation for integer programming. *Mathematical Programming Study*, 2: 82-114.

Goffin, J.L. (1977). On convergence rates of subgradient optimization methods. *Mathematical Programming*, 13: 329-347.

Graves, G.W., McBride, R.D., Gershkoff, I., Anderson, D. and Mahidhara, D. (1993). Flight crew scheduling. *Management Science*, 39:736-745.

Held, M., Wolfe, P. and Crowder, H.P. (1974). Validation of the subgradient approach. *Mathematical Programming*, 6: 62-88.

Herer, Y. and Levy, R. (1997). The metered inventory routing problem, an integrative heuristic algorithm. *International Journal of Production Economics* 51: 69-81.

Hoffman, K.L. and Padberg, M. (1993). Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39: 657-682.

Hooker, J. (2000). *Logic-based methods for optimization*. John Wiley & Sons, New York, USA.

Hunting, M. (1998). *Relaxation Techniques for Discrete Optimization Problems.* Dissertation, University of Twente, The Netherlands.

ILOG (2004). Software company, developer of Cplex. http://www.ilog.com.

Karp, R.M. (1972). Reducibility among combinatorial problems. In Miller, R.E. and Thatcher, J.W. (eds.) *Complexity of Computer Computations*. Plenum Press, New York.

Laporte, G., Gendreau, M., Potvin, J.Y. and Semet, F. (2000). Classical and modern heuristics for vehicle routing problem. *International Transactions in Operational Research,* 7:285-300.

Le Blanc, H.M., Van Krieken, M.G.C., Fleuren, H.A. and Krikke, H.R. (2004A). Collector Managed Inventory, a proactive planning approach to the collection of liquids coming from end-of-life vehicles. CentER Discussion Paper 2004-22, Tilburg University, The Netherlands.

Le Blanc, H.M., Van Krieken, M.G.C., Krikke, H.R. and Fleuren, H.A. (2004B). Advanced planning concepts in the closed-loop container network of ARN. CentER Discussion Paper 2004-103, Tilburg University, The Netherlands. To appear in OR Spectrum.

Marsten, R.E. (1974). An algorithm for large scale set partitioning problems. *Management Science*, 20: 774-787.

Mingozzi, A., Boschetti, M.A., Ricciardelli, S. and Bianco, L. (1999). A set partitioning approach to the crew scheduling problem. *Operations Research*, 47: 873-888.

Müller, T. (1998). Solving set partitioning problems with constraint programming. *Proceedings of the Sixth International Conference on the Practical Application of Prolog and the Fourth International Conference on the Practical Application of Constraint Technology (PAPPACT),* 313-332.

Nawijn, W.M. (1987). Optimizing the performance of a blood analyzer: Application of the set partitioning problem. Memorandum 626, University of Twente, The Netherlands.

NEA (2004). *Kostencalculaties in het beroepsgoederenvervoer over de weg, prijspeil 1-1-2004*. Rijswijk, The Netherlands.

OR-library (2004). Collection of test data sets for a variety of Operations Research problems. http://www.brunel.ac.uk/depts/ma/research/jeb/info.html.

Papadimitriou, C.H. and Steiglitz, K. (1982). *Combinatorial Optimization, Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, New Jersey, USA.

Paragon (2004). Software company, developer of AIMMS. http://www.aimms.com.

Pierce, J.F. and Lasky, J.S. (1973). Improved combinatorial programming algorithms for a class of all-zero-one integer programming problems. *Management Science*, 19: 528-543.

Polyak, B.T. (1967). A general method of solving extremum problems. *Soviet Mathematics Doklady*, 8: 593-597. English translation.

Press, W.H., Vetterling, W.T. and Teukolsky, S.A. (2002). *Numerical recipes in C++.* Cambridge University Press, New York, USA.

Ryan, D.M. (1992). The solution of massive generalized set partitioning problems in aircrew rostering. *Journal of the Operational Reseach Society,* 43: 459-467.

Ryan, D.M. and Falkner, J.C. (1988). On the integer properties of scheduling set partitioning models. *European Journal of Operational Research,* 35:422-456.

Savelsbergh, M.W.P. (1994). Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal of Computing,* 6: 445-454.

Schreurs, R.P. (2004). *Voorspelbaarheid van de inzameling van materialen afkomstig van autowrakken en de impact op de distributieplanning*. Master thesis, Tilburg University, the Netherlands.

Schrijver, A. (1999). *Theory of linear and integer programming*. John Wiley & Sons, Chichester, Great Britain.

Silver, E.A., Pyke, D.F. and Peterson, R. (1998). *Inventory management and production planning and scheduling*. John Wiley & Sons, New York, USA.

Simchi-Levi, D., Kaminsky, P. and Simchi-Levi, E. (2000). *Designing and managing the supply chain*. McGraw-Hill, Boston, USA.

Snowflake (2004). Description of the Bill's Snowflake puzzle. http://www.johnrausch.com/PuzzleWorld/puz/bills_snowflake.htm.

Toth, P. and Vigo, D. (2002). The vehicle routing problem. *SIAM monographs on discrete mathematics and applications,* Society for Industrial and Applied Mathematics.

Van Burik, A.M.C. (1998). Retourstroomproblematiek van materialen afkomstig uit autowrakken. In: Van Goor AR, Flapper SDP, Clement C (eds) *Handboek reverse logistics*. Samson Bedrijfsinformatie, Alphen aan den Rijn, The Netherlands, F3600:1–12.

Van Krieken, M.G.C., Fleuren, H.A. and Peeters, M.J.P. (2003). Problem reduction in set partitioning problems. CentER Discussion Paper 2003-80, Tilburg University, The Netherlands.

Van Krieken, M.G.C., Fleuren, H.A. and Peeters, M.J.P. (2004). A Lagrangian relaxation based algorithm for solving set partitioning problems. CentER Discussion Paper 2004-44, Tilburg University, The Netherlands.

Wedelin, D. (1995). The design of a 0-1 integer optimizer and its application in the Carmen system. *European Journal of Operational Research,* 87: 722-730.

Winston, W.L. (1994). *Operations Research, Applications and Algorithms.* Duxberry Press, Belmont, California, USA.

Yan, S. and Chang, J. (2002). Airline cockpit crew scheduling. *European Journal of Operational Research*, 136: 501-511.

Samenvatting

Dit proefschrift beschrijft een oplossingsalgoritme voor het set partitioning probleem. Dit algoritme is geïmplementeerd in een solver genaamd LaRSS. Het vraagstuk van het set partitioning probleem is om, gegeven een aantal deelverzamelingen van een basis verzameling en kosten behorende bij deze deelverzamelingen, een exacte bedekking van de basis verzameling te vinden tegen minimale kosten. Veel praktijkproblemen, zoals bijvoorbeeld voertuigplanningsproblemen en locatie-allocatie analyses, kunnen geformuleerd worden als een set partitioning probleem en met behulp van LaRSS worden opgelost. In de literatuur zijn meerdere oplossingsalgoritmen voor set partitioning problemen beschreven, waarbij alle succesvolle algoritmen gebruik maken van solvers voor het oplossen van lineaire programmering (LP) problemen om ondergrenzen van het probleem te berekenen. Het doel van ons onderzoek is dan ook om te bepalen of een branch en bound algoritme, gebaseerd op Lagrange relaxatie, zonder gebruik te maken van externe LP solvers, dezelfde prestaties kan leveren als de succesvolle LP-gebaseerde algoritmen die de afgelopen decennia in de literatuur beschreven zijn. LaRSS is het resultaat van dit onderzoek.

Een van de belangrijkste onderdelen van LaRSS is de verzameling probleemreductie-technieken die gebruikt wordt. Probleemreductie-technieken hebben als doel het reduceren van de oplossingstijd van een bepaald probleem door het verminderen van het aantal variabelen of voorwaarden van een probleem. In LaRSS zijn vijf verschillende probleemreductie technieken geïmplementeerd, welke beschreven zijn in Hoofdstuk 2. Bijzonder effectief is de rijcombinatie techniek, welke niet eerder in de literatuur beschreven is. Deze techniek wijkt af van andere technieken, omdat zij een kleine toename van het aantal variabelen toestaat om een reductie in het aantal voorwaarden te bewerkstelligen. Experimenten op onze test verzameling van set partitioning problemen laten zien dat de probleemreductie technieken de oplossingstijd kan reduceren met 80%.

Voor het bepalen van een ondergrens voor het set partitioning probleem gebruiken we de Lagrange relaxatie. Bij een Lagrange relaxatie van het set partitioning probleem worden de gelijkheidsvoorwaarden van het probleem losgelaten waarna het probleem kan worden opgelost met een iteratief algoritme, de subgradiënt methode genoemd. Hoofdstuk 3 beschouwt en vergelijkt verschillende van deze subgradiënt methodes. In LaRSS is een variant op de bekende “convergent

series" methode van Goffin (1977) geïmplementeerd, welke zeer goed blijkt te werken voor set partitioning problemen. De resultaten zijn, zowel in kwaliteit van de oplossing als in rekentijden, vergelijkbaar met de resultaten van de bekende solver CPLEX voor de lineaire programmering relaxatie.

Voor het vinden van een bovengrens is er een eenvoudige, snelle "greedy" primale heuristiek opgenomen in LaRSS. Deze is ontwikkeld door de auteurs. De uiteindelijke oplossing wordt gevonden met behulp van een branch en bound algoritme. Hoofdstuk 5 bespreekt en vergelijkt verschillende branch en bound algoritmen. In LaRSS is een branch en bound algoritme geprogrammeerd dat gebaseerd is op een dynamische volgorde van de rijen van de coëfficiëntenmatrix.

De prestaties van LaRSS op de test verzameling van 60 set partitioning problemen zijn gemeten met CPLEX als referentiepunt. Voor 46 gevallen, ofwel 77%, geldt dat LaRSS de optimale oplossing sneller vindt dan CPLEX. Hoewel LaRSS meestal veel sneller is, zijn er twee problemen waarvoor niet binnen 24 uur een oplossing gevonden wordt, terwijl CPLEX in alle gevallen binnen 7 minuten klaar is. In het algemeen kunnen we zeggen dat LaRSS in het merendeel van de gevallen sneller is dan CPLEX in het oplossen van set partitioning problemen, terwijl CPLEX robuuster is over de gehele verzameling van set partitioning problemen. Dit inzicht wordt bevestigd door verschillende tests op andere verzamelingen van set partitioning problemen.

Het verschijnsel dat de prestaties van LaRSS achter blijven op een kleine verzameling set partitioning problemen kan verklaard worden door drie belangrijke voordelen van methodes die gebaseerd zijn op lineair programmeren (LP). Het eerste voordeel is dat de informatie over de LP oplossing gebruikt kan worden om tijdens de branch en bound snel steeds nieuwe ondergrenzen te berekenen, waardoor veel eerder bekend kan worden dat het geen zin heeft een bepaalde tak van de branch en bound boom verder te onderzoeken. Het tweede voordeel is dat door toepassing van LP tijdens branch en bound veel eerder vastgesteld kan worden dat een gedeeltelijke oplossing niet meer uitgebouwd kan worden naar een toegelaten oplossing van het totale set partitioning probleem. Het derde voordeel is dat met behulp van informatie over de LP oplossing extra voorwaarden, snedes, kunnen worden bepaald die kunnen helpen het probleem sneller op te lossen. Vooral als een probleem een ingewikkelde kostenstructuur heeft, waarbij veel oplossingen met kleine kostenverschillen bestaan, hebben LP-gebaseerde methoden profijt van deze voordelen en zullen ze beter werken dat algoritmen gebaseerd op Lagrange relaxatie, zoals LaRSS.

De waarde van LaRSS is aangetoond met behulp van twee cases die praktijkproblemen van het bedrijf Auto Recycling Nederland (ARN) beschrijven. Voor beide cases zijn simulatiemodellen gebouwd waarin de planning van vrachtwagens over een lange tijdspanne worden nagebootst. In elke periode wordt een planningsprobleem opgelost door het genereren van een groot aantal mogelijke routes en het kiezen van de beste verzameling routes door het oplossen van een set

partitioning probleem. In de eerste case zijn er 14,172 set partitioning problemen gegenereerd en opgelost. In 15 van deze problemen is de optimale oplossing niet gevonden in tien minuten en is de beste oplossing tot dan toe genomen. Deze oplossing is gemiddeld 3% hoger dan het optimum. De gemiddelde oplossingstijd over alle 14,172 problemen is gelijk aan 1.3 seconde. Voor de tweede case zijn er 12,720 set partitioning problemen gegenereerd en opgelost. Van deze verzameling zijn er 635 niet opgelost binnen vijf minuten. De beste oplossing gevonden binnen deze vijf minuten voor deze problemen is slechts 0.14% verwijderd van het optimum. Door CPLEX worden er 92 problemen niet opgelost binnen vijf minuten. Als we de rekentijden van beide solvers vergelijken, zien we dat LaRSS in het merendeel van de gevallen, 69%, het snelste, maar dat er voor LaRSS veel meer uitschieters in rekentijden zijn dan voor CPLEX. Zo ligt de rekentijd van LaRSS in 88% van de gevallen binnen een factor 10 van de beste solver, terwijl dat voor CPLEX geldt in 96% van de gevallen. Ook voor deze problemen geldt dus dat LaRSS meestal sneller is, maar dat CPLEX robuuster is in het oplossen van de gehele verzameling.

Tenslotte is bekeken of we de methoden gebruikt in LaRSS kunnen uitbreiden voor het oplossen van meer problemen. Hierbij hebben we gekeken naar gemengde set packing / set partitioning problemen en naar set partitioning problemen met meer algemene nevenvoorwaarden. Uit experimenten is gebleken dat de gemengde set packing / set partitioning problemen sneller opgelost kunnen worden door ze om te schrijven naar pure set partitioning problemen en LaRSS te gebruiken. Voor de set partitioning problemen met meer algemene nevenvoorwaarden blijkt dat de uitgebreide versie van LaRSS ook goed presteert, en meestal beter dan CPLEX. Hierbij dient wel te worden opgemerkt dat de methodes geschikter zijn voor gelijkheidsvoorwaarden dan voor ongelijkheidsvoorwaarden. Algemene conclusies over de prestaties van deze uitgebreide solver zijn echter moeilijk te trekken, aangezien de experimenten zich hebben beperkt tot problemen met een klein aantal nevenvoorwaarden en gegenereerde problemen.

About the author

Maaike van Krieken was born on January 3, 1979, in ‘s-Hertogenbosch, were she lived until 2001. From 1991 until 1997 she attended the gymnasium (V.W.O) at the Jeroen Bosch College in ‘s-Hertogenbosch.

From 1997 until 2001 she studied econometrics at Tilburg University. During the study she specialized in operations research and she graduated cum laude in August 2001. Her master thesis dealt with medium-term capacity planning problems and was the result of an internship at ORTEC consultants in Gouda.

Directly after her study, she started working as a PhD-student at Tilburg University in the Operations Research group of CentER Applied Research, under supervision of Hein Fleuren and René Peeters. Her research resulted in this thesis and several papers and presentations. As of October 2005, she works as software engineer route planning at Siemens VDO Trading B.V. in Eindhoven.